1993-09

Design and implementation of a query editor for the Amadeus system

Cince, Turgay

Monterey, California. Naval Postgraduate School

REPORT DOCUMENTATION PAGE

PORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
CURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
CLASSIFICATION/DOWNGRADING SCHEDULE	
FORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
ME OF PERFORMING ORGANIZATION puter Science Dept. l Postgraduate School / 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
DRESS (City, State, and ZIP Code) erey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
ME OF FUNDING/SPONSORING GANIZATION / 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
DRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

LE (Include Security Classification)
sign and Implementation of a Query Editor for the Amadeus System.

RSONAL AUTHOR(S)
Lieutenant Turgay Çinçe, Turkish Army

YPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
er's Thesis	FROM 10/92 09/93	93 Sep.	164

PTATION The views expressed in this thesis are those of the author and do not reflect the official policy or ion of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
LD	GROUP	SUB-GROUP	OBJECT-ORIENTED PROGRAMMING, DATA FLOW QUERY, PROGRAPH, VISUAL PROGRAMMING, DATABASE SYSTEMS, SQL, RELATIONAL MODEL, RELATIONAL CALCULUS, RELATIONAL ALGEBRA

STRACT (Continue on reverse if necessary and identify by block number)

e side effect of the proliferation of relational databases within a single organization is that sharing of data to s a global information base is difficult. People erroneously assume that since almost all of the commercially able RDBMSs support the *Structured Query Language (SQL)*, sharing of data is easy. Unfortunately, currently able systems only support a specific dialect of SQL.

e Amadeus front-end system overcomes the data-sharing problem. With the Amadeus front-end system, database can use one common language called *Dataflow Query Language (DFQL)* to access heterogeneous RDBMSs. A / specified in DFQL is correctly translated into a SQL dialect that the connected RDBMS recognizes. With this -end approach, the user can access data from multiple databases by writing a single DFQL query, instead of ng multiple SQL queries. A prototype query builder is reimplemented using an object-oriented design. This ponent of Amadeus interacts with the user for creating DFQL queries. Adding a connection to a new SQL-based

STRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
NCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
AME OF RESPONSIBLE INDIVIDUAL ssor C. Thomas Wu	22b. TELEPHONE (Include Area Code) (408) 656-3391	22c. OFFICE SYMBOL CS/Wq

RM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE: UNCLASSIFIED

[19] Continued:

RDBMS requires minimum modification to the code, due to the object-oriented implementation of the query bu
This object-oriented implementation allows the smooth integration of the additional features of the query editor
the older version of Amadeus.

Approved for public release; distribution is unlimited

Design and Implementation of a Query Editor for the Amadeus System.

by

Turgay Çinçe
First Lieutenant, Turkish Army
BS, Turkish Land Forces Academy, Ankara- Turkey, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1993

ABSTRACT

One side effect of the proliferation of relational databases within a single organization is that sharing of data to access a global information base is difficult. People erroneously assume that since almost all of the commercially available RDBMSs support the *Structured Query Language (SQL)*, sharing of data is easy. Unfortunately, currently available systems only support a specific dialect of SQL.

The Amadeus front-end system overcomes the data-sharing problem. With the Amadeus front-end system, database users can use one common language called *Dataflow Query Language (DFQL)* to access heterogeneous RDBMSs. A query specified in DFQL is correctly translated into a SQL dialect that the connected RDBMS recognizes. With this front-end approach, the user can access data from multiple databases by writing a single DFQL query, instead of writing multiple SQL queries. A prototype query builder is reimplemented using an object-oriented design. This component of Amadeus interacts with the user for creating DFQL queries. Adding a connection to a new SQL-based RDBMS requires minimum modification to the code, due to the object-oriented implementation of the query builder. This object-oriented implementation allows the smooth integration of the additional features of the query editor into the older version of Amadeus.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF QUERIES

ACKNOWLEDGEMENTS

I sincerely thank all of the people who assisted me in the conception and implementation of this thesis.

I would like to acknowledge for the special discussions on Object-oriented design to First Lieutenant Mustafa Eser, Turkish Army, for the precious comments and helps on Macintosh environment to Lieutenant Commander James Phillip Hargrove, United States Navy, and for the precious helps for editing my chapters to Lieutenant Commander Steve Sellner, United States Navy.

Finally, I wish to thank to my wife Pinar for her patience and sacrifice in supporting me in this endeavor. Without her constant love and support, none of this would have been possible.

I. INTRODUCTION

Improvement in relational database management systems (RDBMS) in recent years paves the way for large relational database applications. Since the relational model was first introduced by E.F. Codd in 1969, many companies have used it in a variety of software packages. IBM invented a manipulation language (to write queries) called Structured Query Language (SQL) in 1974. Although ANSI and ISO have established standards of SQL, each vendor supports its own dialect of SQL. When different vendor RDBMSs are required to work together in order to share data, as in a federated RDBMS, an interoperability problem occurs, when a dialect of one vendor's RDBMS cannot be recognized by another.

To solve this problem, a common query language must be used. Using this common query language, the end user can write transactions and use individual translators to convert it to the corresponding dialect of SQL. This implementation can work as a front end, establishing connections between different RDBMS using their individual dialect of SQL, and can solve the problem of interoperability in a federated RDBMS.

A. AMADEUS[1]

Amadeus is an object-oriented implementation of a prototype, which serves as a front end for the end user and provides interoperability between different RDBMSs. Amadeus uses *Data Flow Query Language* (DFQL)[2][Clark91] as a common language for the transactions. It is implemented in the *Apple Macintosh™* environment using an object-oriented language named *Prograph*[3][TGS88a][TGS88b][TGS91].

(1) Amadeus is a prototype developed by several students and continued by myself under advisement of C. Thomas Wu, Prof., Computer Science Department, Naval Postgraduate School, Monterey, CA.
(2) DFQL was implemented by Gard J. Clark as his thesis work in N.P.S. (discussed further in Chapter III)
(3) Prograph is a trademark of The Gunakara Sun Systems, Ltd.

The main goal of this prototype is to provide an alternate query language which will eliminate the differences between RDBMSs in a federation caused by different dialects of SQL. Our implementation includes one back end, the *Oracle*(4) RDBMS, which is available in the Macintosh environment. Connectivity in the federation is maintained by means of each RDBMS' individual dialect of SQL. The user cannot use data types or aggregate functions that are not supported by the connected back-end, and this feature is enforced by Amadeus.

Because of the object-oriented design and implementation of this prototype, it has all the capabilities and benefits of object-oriented programs, including extensibility, flexibility and maintainability. For instance, if to add another RDBMS to the federation, the classes of the back-end RDBMS have to be included with specific feature definitions and methods providing interoperability with the front end. This simple process has the merits of *polymorphism* of object-oriented language. In other words, we do not need to worry about which class methods must be called according to the newly included class, it is done automatically by polymorphism. As a result, the number of back-ends in Amadeus can be increased very easily.

B. THESIS OVERVIEW

Chapter II provides a discussion of available query languages for RDBMSs and the merits and shortcomings of these query languages. The main query language of relational model SQL is discussed in detail, and the difficulties of this language are explained to indicate the need for an easy-to-use common query language. In Chapter III, the DFQL is explained in detail, which is implemented in Amadeus as a solution to the problems of multiple SQL dialects. Examples are given to enhance the understanding of ideas based on an sample database in Appendix A.

The features, pros and cons, and conventions of Amadeus are explained in Chapter IV. Implementation details, such as object-oriented design, class hierarchies, and drawing

(4) Oracle is trademark of Oracle Corporation.

conventions are explained in Chapter V. Chapter VI provides a summary of the research, and gives suggestions for future work.

As mentioned above, Appendix A provides a sample database used for queries in this thesis. Appendix B describes Prograph, the programming language used to develop Amadeus. Appendix C provides the major source code of classes, attributes, and methods used in Amadeus.

II. COMPARISION OF QUERY LANGUAGES FOR RDBMSs

In this chapter, to stress the importance of DFQL, we compare the query languages which can be used with RDBMSs. The relational model(1) is based on a table structure, where relations between tables are established by the *foreign keys*. Therefore, all query languages depend on the connections made through the use of foreign keys to use the relationships between the tables. Query languages for this model can be classified into two general categories: *line oriented* and *visual oriented.*

A. LINE ORIENTED QUERY LANGUAGES

Because of their nature, line-oriented queries can be written using text editors. We can divide this category into three subclasses: relational algebra-based, relational calculus-based and a combination of both.

1. Relational Algebra

In *relational algebra*-based query language, the user specifies a sequence of relational operations to be performed on the tables of his schema to produce the desired result. In Query 2.1, there are three lines which are sequenced with one another until the result is determined. The user can assign temporary names to the result of a previous line to use as an input to the current line. This query language is a procedural type language which is very similar to the *data flow query language* discussed in Chapter III.

```
DEPT_EMPLS <---- (DEPARTMENT *DNUMBER=DNO EMPLOYEE)

RESEARCH_DEPT_EMPLS <----- δ DNAME="Research"
(DEPT_EMPLS)
```

Query 2.1 Find the name and address for all employees who work for the "Research" department.

(1) An example of a relational model is provided in Appendix A.

The main operations of this query language are project, select, and join. Relational algebra-based query language also includes set operations like union and intersection.

2. Relational Calculus

In this type of query language, the user provides a predicate calculus expression which defines the characteristics of the tuples to be retrieved. Tuple variables are used to make the logical connections between separate instances of relations being joined. In Query 2.2, two tables are joined by the common attribute SSN and an existential quantifier is used to retrieve the existing tuples. Since the query only wanted tuples of employees without dependents, the negation of the logical clause is used. As you can see from the query, the free tuple variables are used to reference the attribute names of tables.

```
{e.FNAME, e.LNAME | EMPLOYEE(e) and
( not ( ∃ d DEPENDENT(d)) and e.SSN = d.SSN))}
```

Query 2.2 Find the names of the employees without any dependents.

3. Structured Query Language (SQL)

The third subclass of the query languages is the combination of both relational calculus and relational algebra which includes the nesting capability and block structure established by *SQL*.[2] This language is closer to relational calculus than relational algebra because of its declarative nature. The user specifies the result in one statement rather than a procedural language. SQL queries do not always present the clearest representation to the user. To define a query which has a universal quantification, it must be represented in negative logic and nested queries must be used. As a result, the logical expression to be satisfied becomes quite complicated. Because of the limitations of human nature, the user

(2) SQL was invented by IBM for the relational model. Even though its name is used in many relational query languages, almost all of them have different dialects of SQL which poses a compatibility problem.

can best think of complex problems in sequential fashion, rather than in a declarative fashion of looking at the entire problem at once.

The complexity of the declarative nature of SQL is compensated by embedding SQL queries into a procedural third generation programming language. In this way, the user can take advantage of the features of the host language to accomplish operations that are very difficult to code in the query language. As mentioned above, expressing a universal quantification is very difficult, seen clearly in Query 2.3.

```
SELECT  DNAME
FROM    DEPARTMENT
WHERE   NOT EXISTS ( SELECT *
                     FROM   EMPLOYEE
                     WHERE DNUMBER = DNO AND SALARY <= 30000
                           AND  EXISTS
                                (SELECT *
                                FROM   DEPENDENT
                                WHERE SSN = ESSN )
                     )
```

Query 2.3 Give the department names in which all of its employees have a salary greater than $30,000 and have no dependents.

Existential quantification can be done by using the quantifier **EXISTS** and a nesting select statement. As above, the negative logic, **NOT EXISTS** must be used to express the universal quantification. To complete the query, salary is compared as less than or equal to 30,000 and an **EXISTS** logic is used for dependents.

In SQL, if two relations being used have similarly named attribute columns, a reference must be assigned for those attribute names. This can be done by giving an alias to the relation name *EMPLOYEE (e.g., EMP)* in the **FROM** clause and then similar attribute names may be referenced (i.e., *EMP.SSN).* This process becomes extremely difficult when similar attribute names in a relation are used in nested queries and the user is initially unable to identify which attribute names require aliases.

Two kinds of nesting constructs are used in this query language. One is used in Query 2.3, and the other uses the **IN** operator and a nested select statement. This construct compares attribute names in the outer query with the attribute names returned from the nested select statement. In the previous **EXISTS** construct, at least one tuple must be returned from nested select statement in order to make the **EXISTS** clause true. However, all of these formats create unnecessary complexity and makes the creation of the queries difficult for the user. Although the nesting queries can be translated into their non-nested parts, most SQL optimizers have difficulty translating nested queries.

SQL does not present a simple, clean, and consistent structure to the user and has numerous arbitrary restrictions, exceptions, and special rules. For this reason, this language is called *unorthogonal*(3). An example of an unorthogonal construct in SQL is allowing only a single *DISTINCT* keyword in a select statement even if the select statement contains other nested select statements.

As a result of all these problems, the main query language for RDBMSs cannot be used efficiently by the user. DFQL, which is used in our prototype, solves these problems. DFQL is an efficient query language which can operate with different dialects of SQL in different RDBMSs.

B. VISUAL ORIENTED QUERY LANGUAGES

Visual query languages cannot be written using normal text editors, and require special graphical editors. These types of languages are classified according to their representations. Two categories of visual query languages are *form -based* representation and *entity-relationship*(4) model [Chen76] representation.

---

(3) Orthogonality in a programming language means there is a relatively small set of primitives that can be combined in a relatively small number of ways to build the control and data structures of the language.

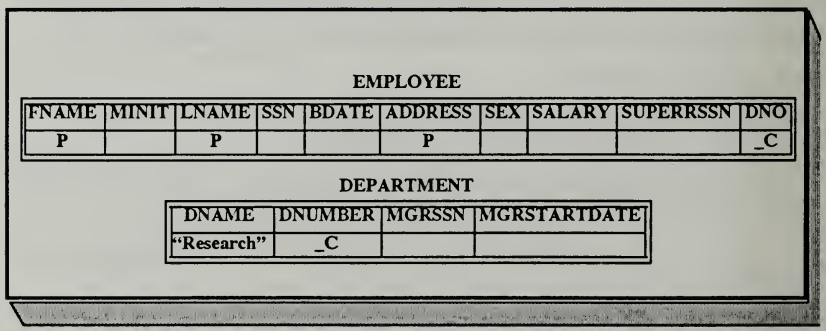
(4) The Entity Relationship Model was introduced by Chen, P. in 1976 as a pictorial conceptual design methodology for the relational model.

1. Form-Based Query Languages

This type is very similar to spreadsheet applications. Most users are already familiar with filling in blank tables or forms; therefore, form-based query languages represent an intuitive language for the user. The main advantage of form-based query languages is they are easy to implement for standard text mode displays. At the time of the creation of these languages, hardware limitations prevented implementing more complex query languages like DFQL. Four types of form-based query languages are discussed in this section, namely, *Query By Example, Summary Table By Example, A Query Language*, and *Relational Calculus/Sets*.

a. Query By Example (QBE)

QBE, developed by IBM in 1976, is the first example of query languages of this type. The user gets a form which represents the attribute names of a given table and types example values into columns which belong to specific attributes of that table. The DBMS then returns the tuples that match the example values provided by the user. As seen



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERRSSN	DNO
P		P			P				_C

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
"Research"	_C		

Query 2.4 This is the representation of (Query 2.1 Find the name and address for all employees who work for the "Research" department.) as in QBE.

in Query 2.4, two tables involved in this query are connected by a variable "_C", according to their primary and foreign keys. "*Research*" is entered for the department name to select the tuples. After selecting the specific tuples, the DBMS retrieves only the attributes which

have a "P" written in its column to indicate that those values will be printed as a result. As in the relational algebra-based query language, QBE uses *free domain variables* to connect the tables to each other. Specific values other than equality can be entered by inserting "<, >, /=" symbols in front of the values entered. For more complex expressions, a separate *Condition Box* can be used to make conditions more explicit.

QBE had great success among users when it was created, because of its user friendly nature. But, as the complexity of the query grows, it becomes less and less useful and it cannot express universal or existential quantification. Therefore, it is not *relationally complete.*

b. Summary Table By Example (STBE)

The representation of STBE is very much like QBE, but it is implemented for a specific area of Statistical Database Management. This language is based heavily on set and aggregation operations. It can deal with relations that have set-valued attributes, summary tables, and aggregate functions using queries that have a hierachical subquery structure. Although there is no implementation of universal quantification, STBE uses set comparison operators to achieve the same result. It can be considered as relationally complete, since it supports all the relational operators. In addition to using a relational model, it has extra capabilities such as supporting summary tables and relations with set-valued attributes.

STBE introduces scoping by allowing nested queries in which table skeletons are placed in nested windows. All the variables used in the table skeletons are bounded by the window. In a nested query, each window contains a subquery and behaves like a function returning an output. The output can be either an output relation skeleton or an output summary table skeleton in the parent window. The outermost window is the root window which returns the result of the query. This nested structure of windows can represent a STBE query as a parse tree. Similar to QBE, a condition box can be used with extra additions of set membership and set comparison.

Although STBE has excellent capabilities, such as powerful aggregation, manipulation of summary tables and relations, and nesting structures, it is a difficult language for novice users who have no knowledge of set theory.

c. A Query Language (AQL)

This query language is implemented for the *AIDE-II* (An Intelligent Database System for End Users) prototype management system which does not incorporate the relational model. Although it is very similar to QBE, it does not have a *join* operator, since the design of the AIDE-II data model does not require it. A user view includes all of the possible relationships in the database. Before a query is defined, a user view must be specified which includes all possible relations to be used. The condition of that specific query can then be defined based on this user view. The disadvantages of AQL include the inability to support the relational model, and the lack of the ability to express joins and universal quantification.

d. Relational Calculus/Sets (RC/S)

RC/S has two graphical implementations very similar to QBE, but it is designed very much like STBE with the ability to use only simple relations. It is a relational calculus-based query language which uses set comparison and set manipulation operators to replace universal quantification in query formulation. The first implementation of RC/S uses nested windows to specify complex queries similar to STBE. The other implementation has the same functionality as the first, but uses hierachical windows to express the nesting concepts. As explained above, *form-based query languages* are designed to be familiar to the user and implemented using current hardware technology.

QBE is the first implemented form-based query language but it is not relationally complete and therefore cannot express some types of queries (i.e., queries using universal quantification). STBE and RC/S attempt to solve this problem while retaining the ease-of-use characteristics of QBE. Even though this problem is solved in these query

languages, these added features detract greatly from the simplicity of the language, since the correct use of set operations requires at least some knowledge of set theory.

AQL eliminates the user-specified join operation from the actual query by requiring a "user view" which unnecessarily separates the query building process into schema manipulation followed by actual query specification. This is certainly not an aid to the user. Additionally, AQL is designed for AIDE-II DBMS which is not a relational model.

2. Entity-Relationship Model Interface

The *Entity-Relationship (ER)* model was introduced by Chen in 1976 and has been extensively used as a high-level conceptual model. The main idea of this model is to illustrate the concepts of entities and relationships in a graphical way in order to enhance understanding of the structure desired for a database.

As illustrated in Figure 2.1, the rectangles represent entities and the diamonds represent relationships between entities. Both entities and relationships may have attributes, represented by connected ovals. Figure 2.1 is intended to specify some of the semantics contained in the sample database.

The ER model is now being used in several query languages rather than just as a conceptual designing model. However, the ER approach has some drawbacks. Although certain relationships are currently specified, it does not necessarily follow that there are no other relationships existing between entities. The intent of the ER model as a query language is to keep the user from worrying about the specific join conditions between entities. However, it tends to force the user to depend on the specified relationships. This is similar to AQL where user views are specified so that all joins are eliminated from the user's view. This can be a benefit to a novice user, but as indicated before, the ability to use a relationship without knowing how it is actually set up increases the chance of syntactically correct queries producing invalid results.

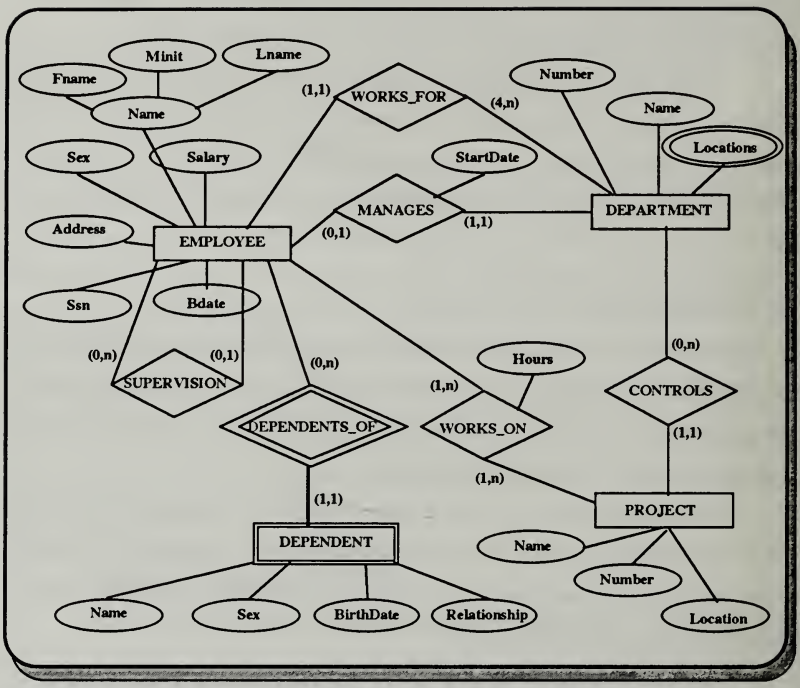


Figure 2.1 ER Diagram [Elmasri89] of sample database in Appendix A

In the original ER model, equi-joins on keys and foreign keys of the entities present no problem. However, if the user desires a theta-join based on some relationship other than equality, even if this theta-join uses the same key attributes as one of the defined relationships, the theta-join would be impossible to perform without adding it as a new relationship to the ER schema.

Five ER-type query languages, namely *Graphical Query Language, Graphical Data Manipulation Language, Query By Diagram, Graphical User Interface for Database Exploration, and GRAphiCal QUery LAnguage* are discussed in the following sections.

a. Graphical Query Language (GQL)

GQL [Andyne91] is a commercial product designed to run as a front end for a user's existing relational DBMS and GQL runs on Macintosh computers. Initially, GQL displays the appropriate ER diagram for the database the user will query. To perform single table or entity queries, the user double clicks on the icon in the ER diagram representing the desired entity. A window with a list of the entity's attributes is then displayed. Attributes may then be selected by applying filtering or sorting conditions to print them on the screen. Queries for specific items like "SSN = 123456789" are formulated with the assistance of GQL's *qualify feature*. The user may connect conditions by boolean operators as well. The information represented by the relationships is accessed by selecting the desired relationship from the screen along with its two adjoining entities. When a query is formed, all of the attributes from both entities are available for qualification and display.

There are several drawbacks to GQL. The possible relationships must be entered by the database administrator(DBA). After that, these relationships are neither changeable nor extensible by the user. When large database schemas have been reduced to third normal form (3NF) with many join conditions, the resulting complete ER diagram may not fit on the screen, causing confusion for the user.

b. Graphical Data Manipulation Language (GDML)

GDML [Czejdo90] uses much of the same type of pictorial representation as the general ER model and GQL. This query language is based on an extended version of the ER model that incorporates various forms of generalization and specialization, including subset, union, and partition relationships. Queries are formed in this language by removing parts of the ER diagram. An editor is provided to allow the user to erase parts of the ER diagram. All of the items in the database represented by the diagram remaining on the screen are then displayed as the result of the query. A method of restriction is provided by allowing the user to place conditions on the attributes in the diagram. Although GDML is based on the ER model for the user interface, as implemented, it runs on top of a

relational DBMS. The GDML entities are simply relations from the underlying database and it's relationships are represented by the database relations containing the appropriate keys from each of the connected entities. As in GQL, the relationships must be established manually as well.

c. Query By Diagram (QBD)

QBD [Angelaccio90] is intended to be a *user friendly* query language based on the ER model which allows the expression of queries with a recursive nature. This language uses the ER diagram as a navigational tool for forming queries. The actual conditions to be satisfied by the query are specified in separate query specification windows.

In this language, the user first selects items of interest from a displayed ER diagram. A window is then opened to place conditions, including recursive ones, on the attributes of that item. By placing two separate entities on either side of the screen, join conditions can be specified between two separate relations. So, by duplicating the same entity on both sides of the screen, recursive queries may be specified.

Two types of windows on each side of the screen are used to accommodate the designer's choice to implement the query formulation process as a series of phases, but these steps seem unnecessarily complex. The formulation of the query in the query condition windows also identifies for the user many options which are not based on the relationships specified in the ER model. But if a query system is to be based on the ER model, then the implementation should stay within the bounds of that model. QBD does not stay within the bounds of the ER model. This anomaly arises from an attempt to provide the flexibility that is missing from the underlying ER model.

d. Graphical User Interface for Database Exploration (GUIDE)

GUIDE [Wong82] has been developed especially to allow browsing meta-data in large databases with many complex relationships. Its design and display methodology are based on the ER model and this query language allows the user to select

a level of detail with which to look at the database. To handle meta-data, entities are organized into a *hierarchical subject directory* and attributes are organized into a *hierarchical attribute directory*. The purpose of these directories is to guide the user to the part of the ER schema that is relevant to him. Also, a facility is provided to *rank* objects according to their expected relevancy to a certain group of users. This ranking is based on the objects expected *importance* in the system. The ranking does not necessarily correspond to the hierarchical organization discussed above, but should reflect the interests of the group of users and the frequency of access to that object by them.

To formulate a query, GUIDE asks the user to first select the level of detail to display for the schema. The ER diagram is then presented at the desired level of detail. Indirect relationships between entities are represented by dotted lines between entities. Next, the attributes of displayed entities and relationships can be examined by selecting the desired object and then examining that selected node. Restrictions can be placed on selected attributes in order to specify the query. The user may select separate portions of the schema to run partial queries, while still maintaining any previous queries. These separate partial queries may then be combined to form a final query.

e. GRAphiCal QUery LAnguage (GRACULA)

GRACULA is implemented by IBM as a graphical language for querying and updating a database. It is based on the definition of a database schema that is presented to the user in the form of ER diagram. The relationships are displayed simply as directed arcs between the entities with the appropriate relationship name attached to the arc. The database schema is displayed in one window while the query is built up in a separate query window. The query window is initially empty. The user selects entities from the schema window which are then displayed in the query window for further manipulation. To formulate the query on the items the user has placed in the query window, the items may be expanded to show their attributes. The attributes are listed in a tabular fashion and restriction conditions can be entered for them somewhat as in QBE. Joins between items

which are unrelated in the schema can be performed by specifying the join attribute from one entity in the other entities value column.

Additional power is added to this language by nesting simple entities and relationships inside various frames. A frame is indicated by a box drawn on the screen which may contain one or more entities and their associated conditions and relationships. These frames are used to specify logical operations such as AND, OR, NAND, or NOR and implication and consequent. The logical operations are scoped over any of the entities and relationships that are contained in their frame. Nesting of operations can thus be performed by nesting frames, providing a clear way of showing the scope of each of the operations. The inclusion of implication and consequent frames is intended to ease the problem of specifying universal and existential quantification. As stated previously, the predicate logic approach for these ideas is not simple.

We have discussed the query languages which are based on ER model representation. Each has its own way of expressing queries while adhering to the ER diagrams for definition of the database. The ER model has a certain advantage in that it can simplify the query and make it easy to understand. Also, the database schema is displayed so the user does not have to memorize the specific relationships between database objects. But, it has the fallowing drawbacks:

- Using the actual schema to define queries (although this is an advantage for ease-of-use) limits the user to predefined relationships that have been coded into schema.
- Most ER systems assume relationships based on the equi-join of keys between entities. This does not take into consideration relationships based on other attributes or on other types of theta-joins.
- The distinction between entities and relationships is not straightforward. For example, in an airline flight, to an accountant it exists as an entity (a concrete object), but to a scheduler, it exists as a relationship between a specific aircraft, aircrew, routing, etc. This lack of concrete distinction could cause problems when queries must be made from a single ER schema by multiple users, each with a different point of view.

In the next chapter we will discuss DFQL implemented in Amadeus, and its advantages and disadvantages.

III. DATA FLOW QUERY LANGUAGE (DFQL)

DFQL is a visual/graphical query language for RDMSs based on a dataflow paradigm. It has all of the capabilities of existing query languages which can be extended by the user by creating new operators from the existing primitive or user-defined operators. DFQL includes aggregate functions in addition to the operators of a relationally complete query language. It has the power of expressing every kind of expression, including universal and existential quantification. The following goals are met by DFQL [Wu91a]:

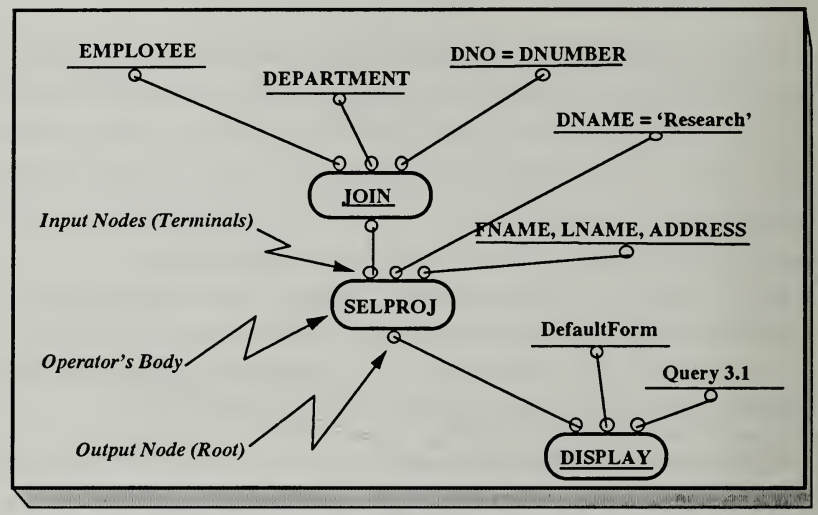
- Employ a fully graphical environment as a user-friendly interface to the database.
- Sufficient expressive power and functionality, including relational completeness.
- Ease-of-use in learning, remembering, writing and reading the language's constructs.
- Consistency, predictability, and naturalness (in both syntax and function).
- Simplicity and conciseness of features.
- Clarity of definition and lack of ambiguity.
- Ability to modify existing queries to form new ones incrementally.
- High probability that users will write error-free queries.
- Operator extensibility(1)

To achieve these goals, DFQL adheres to relational algebra and maintains the requirements of operational closure. It also eliminates the range variables and nesting features used in SQL. The most important feature of DFQL is the ability of the user to treat relations as abstract entities operated on by relational operators. As a result, the user can compose his queries in the realm of relational algebra and does not have to worry about how operations are carried out.

A sample query represented in DFQL form is presented in Query 3.1. Two types of operators are used in DFQL, *primitive* and *user-defined*. Primitive operators' names are identified by underlined texts in Query 3.1. An operator has three parts: *terminals, body, and root*. According to the dataflow paradigm, data flows from the upper operators' roots

(1) Operator extensibility allows the user to create new operators in terms of existing ones, analogous to defining a function in a programming language.

to the lower operators' terminals through the arcs. As soon as all the data on the terminals of an operator are ready, that operator is fired to execute the specific process. In this query,



Query 3.1 Representation of Query 2.1 (Find the name and address of all employees who work for the "Research" department) in DFQL format.

two relations (employee, department) are joined according to their primary and foreign keys. The combined relation flows from the root of the same operator to the first terminal of the user-defined operator named *selproj*. This operator is defined by the user and combines *select* and *project* primitive operators to make the query more readable and easy to use. When this operator is fired, it gets the combined relation, selects the tuples with attributes of the combined relation *DNAME named* 'Research' and then projects the columns of attribute names *FNAME, LNAME, and ADDRESS*, respectively. In DFQL, each query has to have a *display* operator in order to show the resultant tuples to the user in the specified form and with the given title.

In our implementation of DFQL, an operator can be executed only once; iteration or recursion is not permitted. These features can be added within the bounds of Amadeus'

programming tool's capability[2]. Orthogonality is applied to the implementation of DFQL to maintain clarity and lack of ambiguity. The functional paradigm is fully supported by DFQL notation and all DFQL operators implement *operational closure*. In other words, the inputs to the operators are relations or associated textual instructions, and the output from each operator is always a relation. This idea is fundamental to the understanding of large and complex queries. If operational closure is not enforced, some operators give a relation as an output whereas others give some different type of data. This means that every operator must be connected according to its type of inputs/outputs. However, this is very cumbersome when the query being formulated is complex in its own right. Because of operational closure in DFQL, this burden is eliminated and all operators can be connected to each other without any concern of incompatibility.

A. DFQL OPERATORS

In this section we will explain the operators used in DFQL to build queries and provide some examples of their usage in queries. As mentioned previously, DFQL operators are divided into two parts, namely *primitive operators* and *user-defined operators*. User-defined operators are not in the system and must be defined by the user. Primitive operators have direct execution in code without any translation. Primitive operators are categorized as *basic* and *non-basic*.

1. Basic DFQL Primitive Operators

Since this query language is relationally complete, it must have five primary operators: *select, project, union, join,* and *difference*, is illustrated in Table 1. These operators are implemented as a basic set of operators in DFQL. Using these five primary operators, the user can build even more complex operators. Also, a *groupCnt* operator is included as a basic operator for simple aggregation. This operator provides an easy solution to universal quantification problem (discussed later). The SQL representations of the basic operators are included in the table for comparison.

(2) Amadeus' tools for the programmer will be discussed in Chapter IV in detail.

As can be seen from the graphical representation of the operators, other DFQL objects are included in addition to the primary operators. These are *textual objects* which can be fed the conditions, attribute lists, or any alias names to be used in the operator. *Relation* and *form objects* are fed as input to operators during execution of an instance of these objects. The DFQL objects are represented by drawing a line underneath the text. However, the form object is represented by a double line underneath the text. Sending text to the operator as an input is a design decision and does not violate orthogonality or relational completeness of the query language.

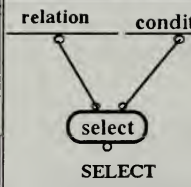
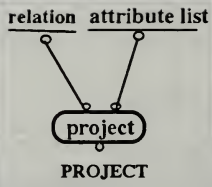
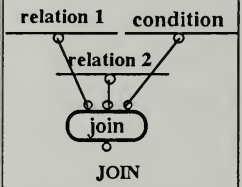
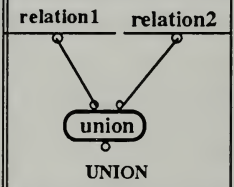
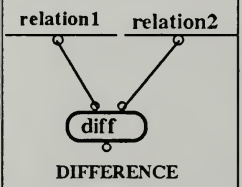
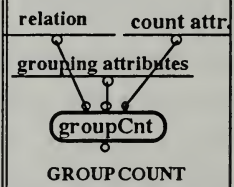
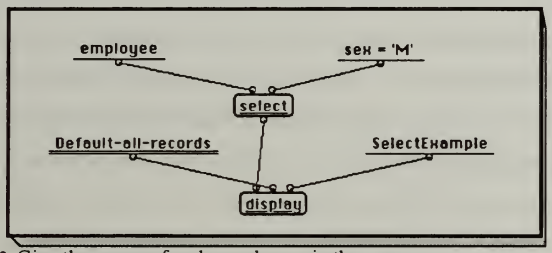
DFQL	SQL	DFQL	SQL
 SELECT	SELECT DISTINCT * FROM relation WHERE condition	 PROJECT	SELECT DISTINCT attribute list FROM relation
 JOIN	SELECT DISTINCT * FROM relation1 r1, relation2 r2 WHERE condition	 UNION	SELECT DISTINCT * FROM relation1 UNION SELECT DISTINCT * FROM relation2
 DIFFERENCE	SELECT DISTINCT * FROM relation1 MINUS SELECT DISTINCT * FROM relation2	 GROUP COUNT	SELECT DISTINCT grouping attribute COUNT(*) count attr. FROM relation GROUP BY grouping attributes

Table 1: BASIC DFQL OPERATORS AND THEIR SQL TRANSLATIONS

a. Select

This operator implements the relational algebra operation of database selection. Its notation in relational algebra is $\delta_{\text{<condition>}}(\text{<relation>})$. It retrieves tuples from the relation which fits the specified condition. After the operation, the relation is reduced in size, containing only the tuples that maintain the condition. There is no alteration to the structure of the relation, so that it is operationally closed. Also, the resultant relation is proper, in that there is no duplicate row. Proper relations will be discussed exclusively in subsequent sections, unless otherwise stated.



Query 3.2 Give the names of male employees in the company.

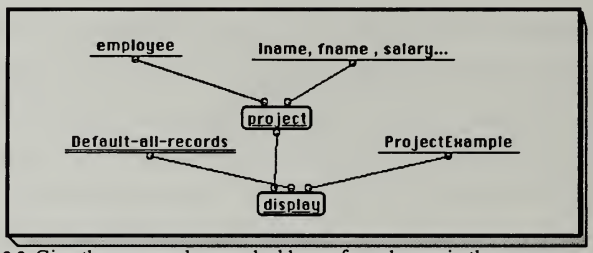
In Query 3.2,[3] a relation named *employee* with the condition of attribute *sex* = *'M'* is given as input to the select operator. The resultant relation flows out while containing only the tuples which match the condition specified. The relation is printed to the screen by the *display* operator (discussed below) with the specified name and form.

b. Project

The notation of this operator in relational algebra is $\Pi_{\text{<attribute list>}}(\text{<relation>})$ which stands for database projection. The attribute list contains the names of attributes to be retrieved from the relation separated by commas. The result of the projection is a proper relation which contains only the columns of specified attribute

(3) All query results in this document are presented in Appendix A.

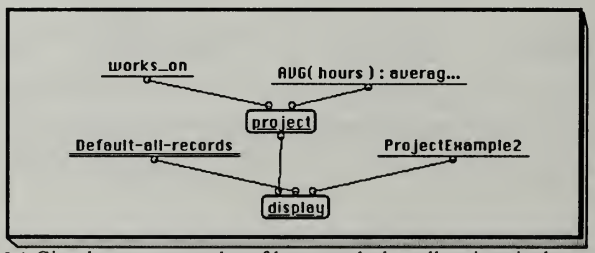
names. In other words, the project operator eliminates the duplicates in the remaining columns, since the key attribute of the relation may not be desired as a result.



Query 3.3 Give the name, salary, and address of employees in the company.

It can be seen from Query 3.3 that the relation *employee* and its attribute list are given as inputs to the operator. A proper relation containing those attributes is then sent to the display operator to be printed on the screen as described before. The project operator can be used to change the attribute name in the relation when required. Instead of inputting an attribute list, an equality condition like "*gpa = grade*" is input to change the attribute *grade* to *gpa*.

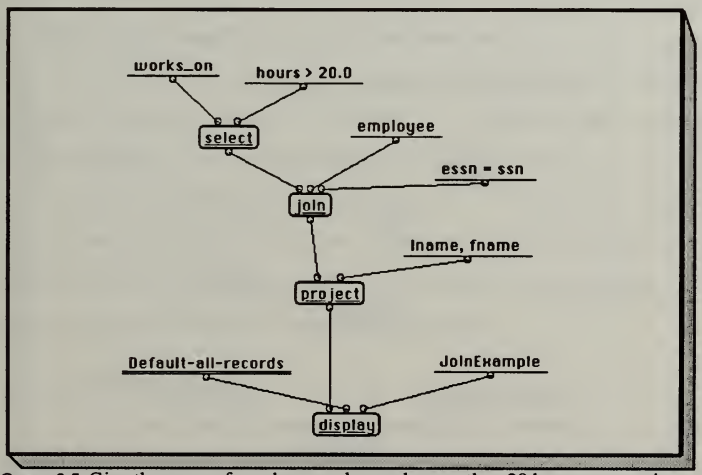
Although DFQL provides complete grouping aggregate functions as separate operators (discussed in the next section), the user can use *cnt, min, avg, max, and sum* aggregate functions without grouping. They affect all tuples in the relation by using the function name in the attribute list of project operator (i.e., "*sum (salary): total of salaries*"). An alias name must be given after a colon to indicate the name of the result. Another example of aggregate function using the same notation is shown in Query 3.4.



Query 3.4 Give the average number of hours worked on all projects in the company.

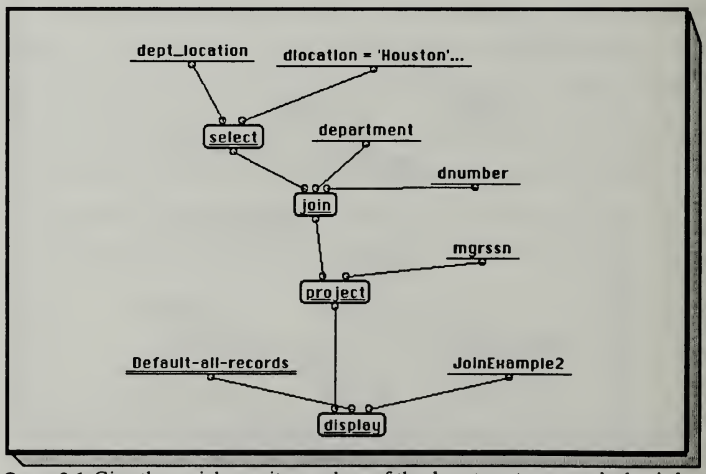
c. Join

The relational algebra notation of this operator is **<relation1>*_{<condition>}<relation2>** implemented as a theta-join in Amadeus. The relation formed by a join operation results in all attributes from both relations combined together as a cartesian product of tuples satisfying the specified condition. The user may not necessarily give any condition, therefore, the join operator becomes a cartesian product. If both of the relations have the same named attribute used in the condition, the order of the relation coming in to the operator is left to right. Since in the join operation there are only two relations involved, the user must pay attention to the order of the attributes. In the translation and communication section of the back-end, Amadeus gives range variables prefixed to similar attributes, since all the manipulation can be done by SQL with the back-ends. In such cases, the user has the option of providing only one attribute name to indicate the likeness of the names and DFQL makes an equality condition for attributes coming from both relations.



Query 3.5 Give the name of employees who work more than 20 hours on a project.

An example of a theta-join is given in Query 3.5. This query shows the employee relation joined with the result of the select operation, applying the condition of key and foreign key attributes of these relations. In order to find the employee names in different relations, selected employee tuples had to be joined with the relation containing the attributes *lname* and *fname*. Since the attribute names in the join condition are not alike, the user does not need to worry about the order of the relations and attribute names in this query. This operation retains all the attributes of the result, therefore, attributes with the same name resulting from the join must be handled differently. Since the relational model does not allow two column names in one table, one of the similar columns can be discarded in the equi-join condition. However, this solution is not always optimum. Hence, the second column name is changed by suffixing a "1" at the end, preventing an equality in all join conditions. This case is represented in Query 3.6 where the column *dnumber* is same in both relations, (i.e., *DEPARTMENT_LOCATIONS and DEPARTMENT).* Since the join condition is an equi-join, one of the dnumber columns may be discarded since they both have same information. In our implementation, the second *dnumber* is changed to



Query 3.6 Give the social security numbers of the department manager's that is located in "Houston".

dnumber1 in the joined relation. As another example, the user may join two relations like *employee* and *dependent* (see Appendix A) with like column names (e.g., *sex).* In this case, it would be improper to discard the second column because they have different information relating to the resultant relation. The option of discarding one of the like columns is a special type of join called a natural join, which is not implemented in our prototype.

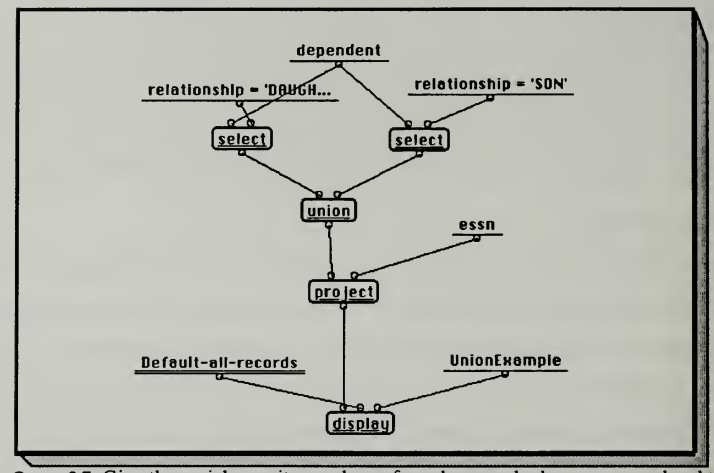
d. Union

The union operator implements the relational algebra operation of union and its notation is **<relation1> U <relation2>**. This operator combines all the tuples from both relations while eliminating duplicates. It does not create a new relation with different structure, which is why both relations must be union compatible. In other words, the number of attributes, their names and types must be the same and in the same order. This rule is valid for all of the set operators used in DFQL. The user may confuse this union operator with mathematics' union operator. According to the mathematics definition of union, the operation takes two sets, eliminates the duplicates and makes another set from the combination. However, as showed here, the DFQL union does not create any new relations other than combining the tuples of both relations.

We have used the union operator in Query 3.7 for union compatible relations coming from select operators, since the select operator does not affect the structure of the relation. Two relations, one containing tuples of employees with a son or sons as dependents, the other containing tuples of employees with a daughter or daughters as dependents are combined by the union operator. Since some employees can have both a son and a daughter, these tuples will exist in the resultant relation. These duplicates will then be discarded by the project operator to make a proper relation.

e. Difference

The relational algebra notation for this operator is **<relation1> - <relation2>**. Relational difference returns as a result a relation that contains all the tuples that occur in **<relation1>** but not in **<relation2>**. In other words, it renames tuples from **<relation1>**



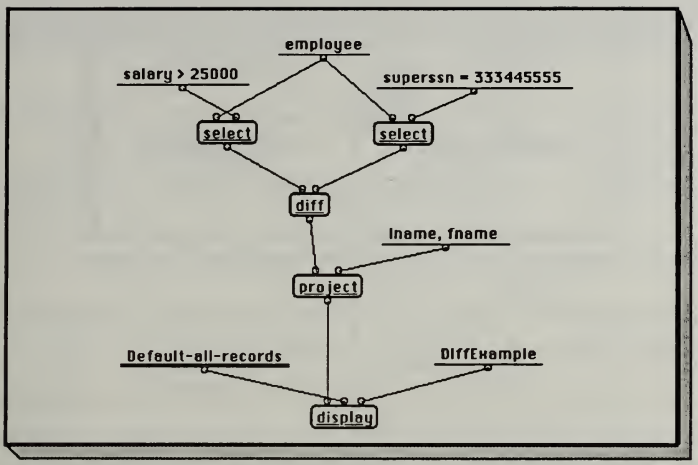
Query 3.7 Give the social security numbers of employees who have a son or daughter as a dependent.

which occur in **<relation2>**. Like the union operator, both relations must be union compatible.

As we can see from Query 3.8, this operator is used to discard the tuples of employees which are under the management of a given supervisor. First, the relation of employees having a salary of more than $25,000 is selected, and then the employees under the management of given supervisor are selected. These two relations are union compatible since they are derived from the same relation. The difference operator removes the tuples from the first relation which exist in the second relation. A project operator then filters the columns related only with the employee names.

f. Group Count

This operator is provided as a primitive operator to provide the user with some simple aggregation capabilities. It is very important for the user to be able to formulate queries involving universal quantification[4]. This operator counts the number of the tuples in a particular grouping specified by the user. It takes a relation, a list of grouping



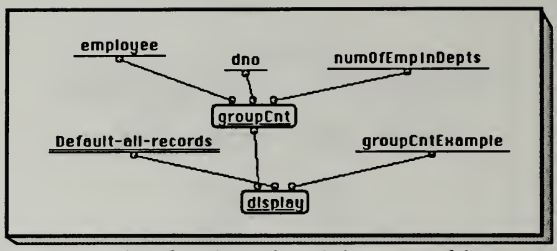
Query 3.8 List the names of employees who have a salary greater than '$25,000' but not under the management of a supervisor who's social security number is '333445555'.

attributes, and an alias name for the result. Grouping attributes can either be one attribute or several attributes, separated by commas. The resultant relation of this operator is a relation with grouping attributes listed in the same order and the result, which is given an alias name. The count result is an integer providing the total number of tuples in that grouping. As can be seen in Query 3.9, the number of employees in each department are counted by giving *dno* as a grouping attribute for the relation *employee*. An alias name is given to be used as the resultant column's name.

2. Non-basic DFQL Primitive Operators

Several other primitive operators have been included in DFQL of Amadeus that can do special operations on relations. These primitives perform low level operations that the user would not include as user-defined operations. However, all of them can be

(4) The solution of universal quantification will be discussed in section B.2. Universal Quantification .



Query 3.9 List the number of employees in each department of the company.

defined from basic primitives as user-defined operators. In Table 2, non-basic primitive operators are compared with their SQL correspondents.

An advantage of creating these operators is to use the built-in functions of the underlying DBMS we are running as a back-end. For example, the *intersection* operator can be defined in terms of the existing *union* and *diff* operator as in the formula $R1 \cap R2) \equiv (R1 \cup R2) - ((R1 - R2) \cup (R2 - R1))$. However, many DBMSs already provide a specific intersect operator and using the intersect operator already provided by the back-end is more efficient. If left to the user to be implemented as a user-defined operator, the advantage of using predefined operators from the back-end is lost. User-defined operators induce only a little overhead to process the operator since it must access its primitive constituents one by one and execute them. This is not a big problem, but when compared to using the operator provided by the back-end, the difference is significant.

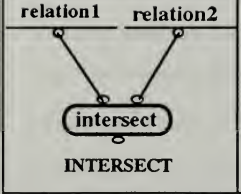
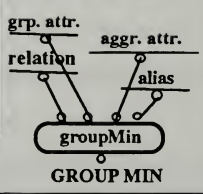
DFQL	SQL	DFQL	SQL
 INTERSECT	SELECT DISTINCT * FROM relation1 INTERSECT SELECT DISTINCT * FROM relation2	 GROUP MIN	SELECT DISTINCT grp.attr.,min (aggr. attr.) FROM relation GROUP BY grp.attr.

Table 2: NON-BASIC DFQL OPERATORS AND THEIR SQL TRANSLATIONS

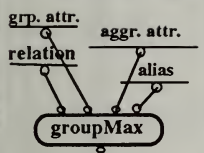
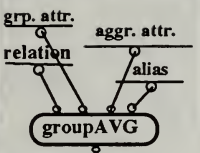
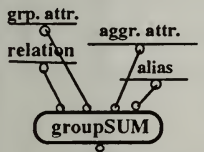
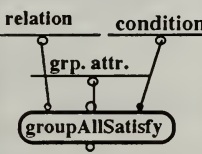
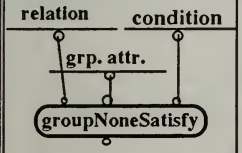
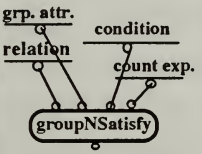
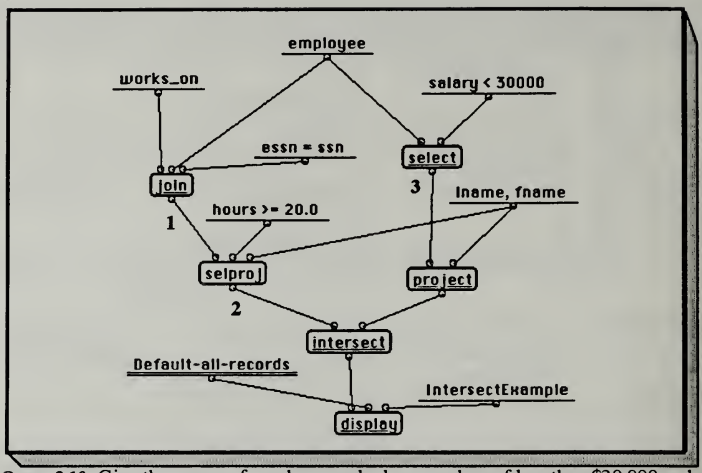
DFQL	SQL	DFQL	SQL
 GROUP MAX	SELECT DISTINCT grp.attr.,**max** (aggr. attr.) **FROM** relation **GROUP BY** grp.attr.	 GROUP AVG	SELECT DISTINCT grp.attr.,**avg** (aggr. attr.) **FROM** relation **GROUP BY** grp.attr.
 GROUP SUM	SELECT DISTINCT grp.attr.,**sum** (aggr. attr.) **FROM** relation **GROUP BY** grp.attr.	 GROUP ALL SATISFY	It is not implemented directly in SQL.
 GROUP NONE SATISFY	It is not implemented directly in SQL.	 GROUP N SATISFY	It is not implemented directly in SQL.

Table 2: NON-BASIC DFQL OPERATORS AND THEIR SQL TRANSLATIONS

a. Intersect

This operator implements the relational algebra operation of intersection with the notation of ***relation*1 ∩*relation*2**. It retrieves tuples which exist in both relations and give the combination as a result relation. The input relations must be union compatible as described for the *union* and *diff* operators. The usage of this operator is explained in Query 3.10, where this operator plays the role of the *AND* conjunction. At *point one*, two relations are joined according to their key and foreign keys (e.g *essn* and *ssn)* in order to manipulate the attributes needed for the query. The tuples of employees who worked more than 20 hours are selected and the named columns are projected by a user-defined operator at *point two*. At *point three*, employees having a salary less than $30,000 is used as the



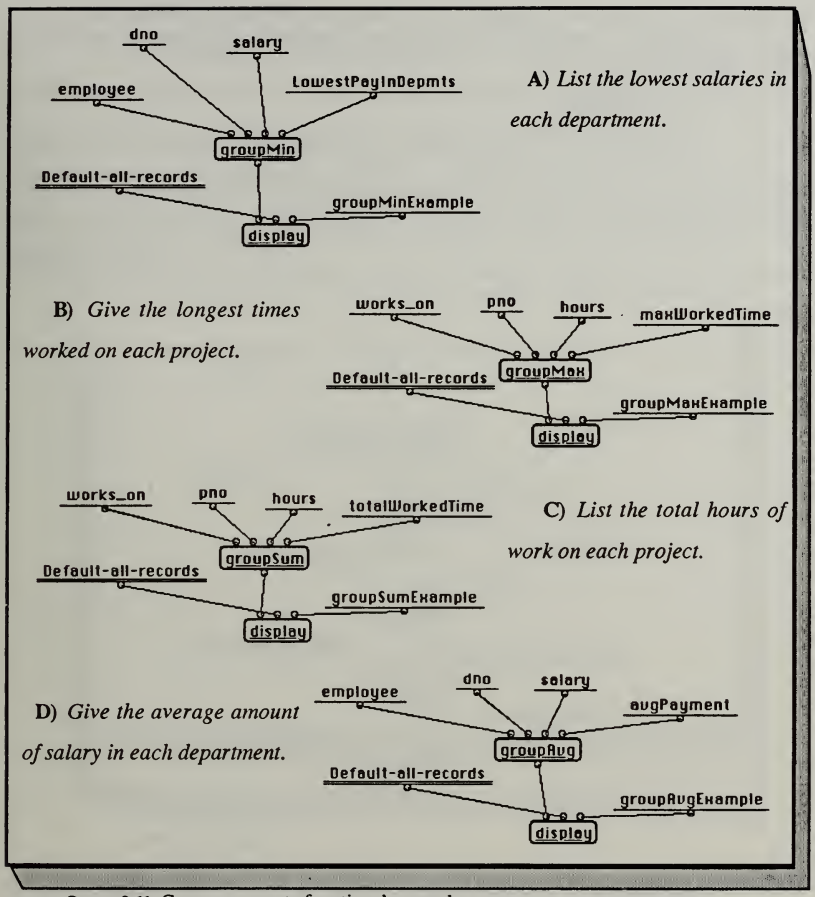
Query 3.10 Give the names of employees who have a salary of less than $30,000 and worked more than 20 hours on any project.

other named condition selected. A *project* operator is used to make the relation union compatible with the previously selected relation at *point two*. Finally, the *intersect* operator combines the tuples which exist in both relations to force both selection conditions.

b. Grouping Aggregate Operators

The rest of the grouping aggregate operators in addition to *groupCnt* are included in the system, to allow the user to take advantage of these functions. These operators cannot be implemented as user-defined operators. *GroupMin*, *groupMax*, *groupSum*, and *groupAvg* are discussed in the following section.

- *GroupMin* finds the minimum value of the specified attribute in the separated sections according to the grouping attributes. It places the grouping attributes and the minimum values of each group in separate columns. The minimum values column is given an alias by the user. Its example is illustrated in Query 3.11 section A. Here the lowest valued salaries are selected for each department from the relation *employee*.
- *groupMax* is similar to the previous operator except it finds the maximum value of the aggregating attribute according to the grouping attribute. An illustration of this



Query 3.11 Group aggregate functions' examples.

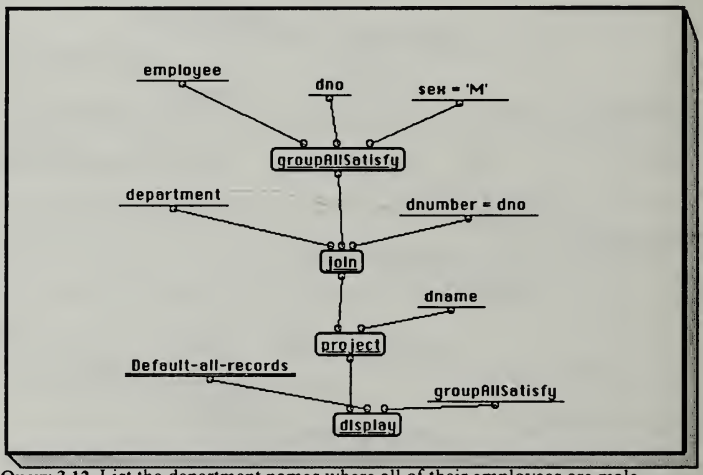
operator is seen in the same query, section B, where the maximum working hour values are selected for each project.

- *GroupSum* is shown in section C of the query, to find the total of hours worked on each project. This operator adds all of the aggregated attribute's values in each section of grouping attributes. The grouping attributes and calculated values are again placed in separate columns. The calculated values column is given an alias by the user.

- *GroupAvg* calculates the average of the given aggregate attribute according to the grouping attribute. In section D, the operator is illustrated finding the average salary for each department.

c. GroupAllSatisfy

This operator is a simple universal quantification included for the user's convenience. It takes a relation and splits the tuples according to the grouping attribute list and then checks all tuples in individual groups according to the specified condition. If all of the tuples satisfy the specified condition then the values of that grouping attribute list are presented.

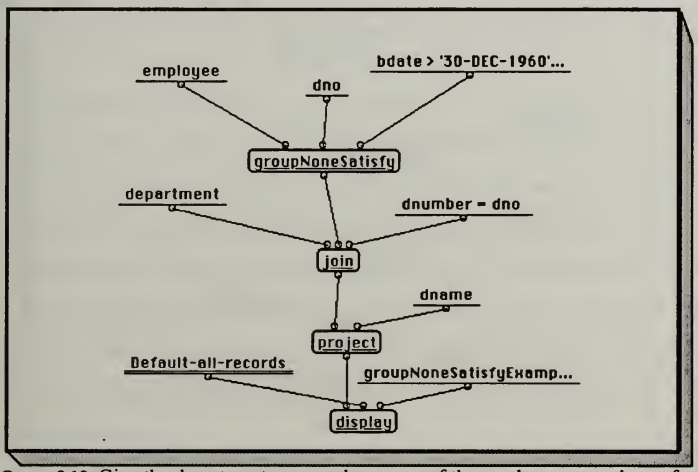


Query 3.12 List the department names where all of their employees are male.

An example of *groupAllSatisfy* is shown in Query 3.12. The condition is specified to find the department names where all of their employees are male. The attribute *dno* is given as a grouping attribute for the relation *employee*. The result from this operator is the number of departments satisfying the specified condition in all tuples. Join and project operators are used to find and project the department names instead of numbers.

d. GroupNoneSatisfy

This operator is the opposite of the *groupAllSatisfy* operator in that it gives the grouping attributes only if none of the tuples satisfy the condition. The notation and usage are the same as the previous operator. This operator is used in Query 3.13, where the department numbers are selected in which none of the employees were born after 1960. As done in the previous query, *join* and *project* operators are used to find the department name instead of number.



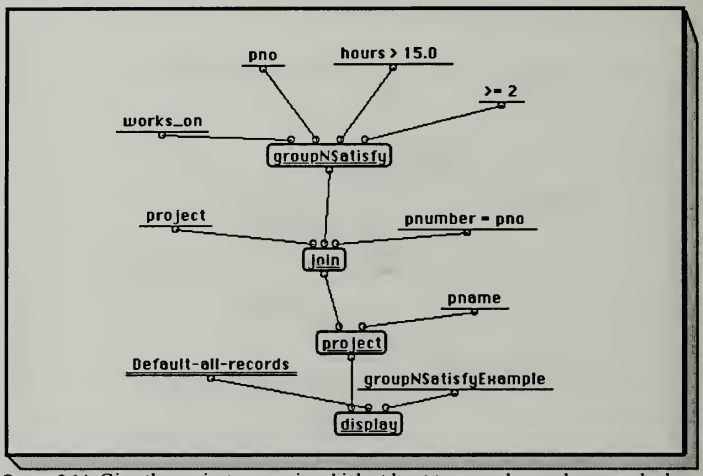
Query 3.13 Give the department names where none of the employees were born after 1960.

e. GroupNSatisfy

This operator takes an input in addition to the other three inputs of *relation, grouping attributes, and condition.* The extra input specifies the number of tuples which must satisfy the condition in order to pass the grouping attributes. Previously discussed operators check the condition for all tuples or for none, but here, the user can specify a middle number and can indicate an operator like *less than*, or *greater than*, to specify which

side of the number will be considered. The result is the same as previously discussed operators that pass the grouping attributes as a resultant relation.

This operator is used in Query 3.14 to find the project names in which at least two employees worked more than the specified hours. The numeric condition (>=) is used to force the requirement *at least* in the query. This query passes the project numbers in which at least two employees have worked more than specified hours. The result is joined with the relation *project* to find the project names instead of passing the project numbers.



Query 3.14 Give the project names in which at least two employees have worked more than 15.0 hours.

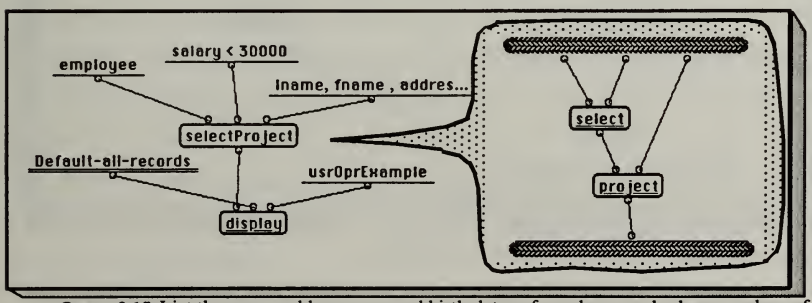
3. User-Defined Operators

These operators give the user flexibility to define his own style of operators and extend the capability of the language according to the user's desires. These operators look like a primitive operator, except its name is written without underlined text, and they can be constructed from available primitives and previously defined user operators as well. User-defined operators can be used in any level of nesting to formulate new operators. This feature does not decrease the power of orthogonality, since every user-defined

operator must be defined from a primitive operator or a previously-defined user operator, which adheres to the principle of orthogonality. Some advantages of these operators are:

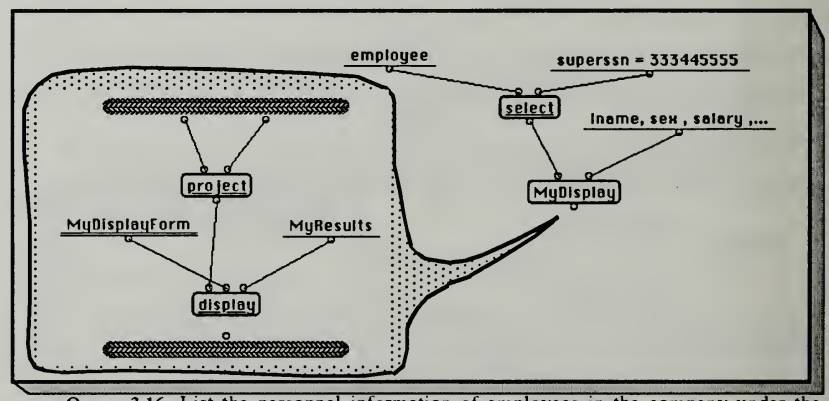
- Gives the user the flexibility to extend the language according his style
- Encapsulates the detail and makes the query more understandable
- Saves space on the screen or in the drawing area
- Allows the use of previously defined and correct portions of complex queries in more than one query easily and correctly while maintaining the complexity in itself
- Enhances the ability to write error-free queries and saves time from debugging queries level by level after construction
- Allows abstraction and encapsulation principles of software engineering in the field of query languages

A user-defined operator is used in Query 3.15, which combines two primitive operators (e.g., *select* and *project).* These two operators are very often used together in queries to select tuples, according to the specified condition, and project the desired columns as a result. Hence, the user can combine these two operators into a user-defined operator and can give it a related name (here *selectProject)* to provide an indication of its purpose. In the example, the desired result is employees who have a salary less then the specified amount and the columns of personnel information as written in the query. The *SelectProject* operator is very useful in this type of query, and if used, provides all of the merits mentioned above.



Query 3.15 List the name, address, sex, and birth dates of employees who have a salary of less than $30,000.

To define a user-defined operator, the user has to decide how many inputs will be used, but as in all other operators, there is only one output from the user-defined operator. The user's operator definition has two bars which stand for *input and output bars*. The user connects the input nodes to his operators. The user is not required to connect anything to the output node. As can be seen in Query 3.16, the user-defined operator may contain a display operator which does not have any output. Here a user may prefer to create an output operator instead of using the primitive operator *display*. Alternatively, the user may prefer to see the results in his created form and he may want to see only specific columns. To do this, the user may use a *project* operator to pass the selected columns and then use the *display* operator with a defined form name (here *MyDisplayForm)* in each query. But this is cumbersome, so the user may instead define the operator only once and use it in any query desired. In Query 3.16, after selecting the employees under the specified management personnel, *MyDisplay* is used to project the personnel information and then display them in the form and with the titles according to the user's desires.



Query 3.16 List the personnel information of employees in the company under the supervision of the manager whose ssn is '333445555'.

4. Display Primitive Operator

As seen from the previous queries, only one *display* operator is used to print the results of the queries to the screen. This operator has three input nodes and no output node, because it does not return a relation after execution. In our implementation, it is required that every query must contain a *display* operator. Using this operator, the first input is the relation to be displayed, the second input is the name of the form object where the data will be displayed, and the last input is an alias to be printed as a title in the resultant form. Since there can be more than one *display* operator (especially while debugging) in the queries, this alias name is needed to distinguish the results. Also, the form object is drawn with double lines to distinguish it from other DFQL parameter objects. Two default form objects are included to show the results. One displays all values as one line for each tuple in the relation, and the other displays one tuple at a time. This feature will be explained in the next chapter in detail. Except for the relation, the other two inputs may be omitted by the user and a default form object and title can be used for convenience.

B. QUERY CONSTRUCTION WITH DFQL

Query construction has been implicitly explained in the query examples so far. Some important features of DFQL query construction are discussed here. DFQL is a complete dataflow diagram (DFD) which adheres all the rules of the DFD paradigm. The operators and objects are connected to each other by lines called dataflow paths and all of the information traverses these paths during execution. Except for operators, DFQL objects do not have any input nodes and can be executed any time. They pass the relation objects, attribute lists, or conditions to allow use by the operators. As soon as all of the input nodes have the required/specified information, an operator can be executed or fired and produce a relation at its output node. This relation can flow to other connected operators, making these operators ready to fire.

Since DFQL query execution does not permit iteration and recursion, each operator can be fired only once. Therefore, in our implementation, query execution can start from

the bottom (from *the display* operator), traversing upward by checking each operator's input nodes. If all the input nodes contain data, the query fires the operator, takes the result and turns back; otherwise, it continues traversing upward in order to get the required data from upper levels. The execution finishes at the starting operator, printing the results for the user.

1. Incremental Queries

The most important feature of DFQL is to allow the user to build queries incrementally. In other words, the user can formulate one portion of the query, check the results, (return back if needed), and continue to build other portions of the query one by one. This gives the user more flexibility during his work, especially when the query is very complex. This prevents the user from loosing himself in the total query and can provide intermediate results in order to proceed with further construction. An incremental query can be divided into two sections, namely *incremental construction* and *incremental execution.*

a. Incremental Construction

Incremental construction is the ability to build the query part by part while determining the results of each part. This is very important when the complexity of the query grows. An example is used to explain this feature.

Three sections can be seen in Figure 3.1, showing phases of incremental construction. These phases depend on the logical portions of the required English statement. For this complex query, the user can construct the query in three phases. First, the user can find the "*department numbers where all employees have a salary greater than a specified amount*" as in section A and check the result for correctness. If the result is not correct, the user can make the correction, and check the intermediate result again. After one condition of the query is satisfied, the user can then move on to section B, which formulates the tuple *"all employees with no dependents*" and the department numbers satisfying the condition are passed. The last section combines these relations according to *the AND*

conjunction to enforce both specified conditions and to obtain the department names and display them.

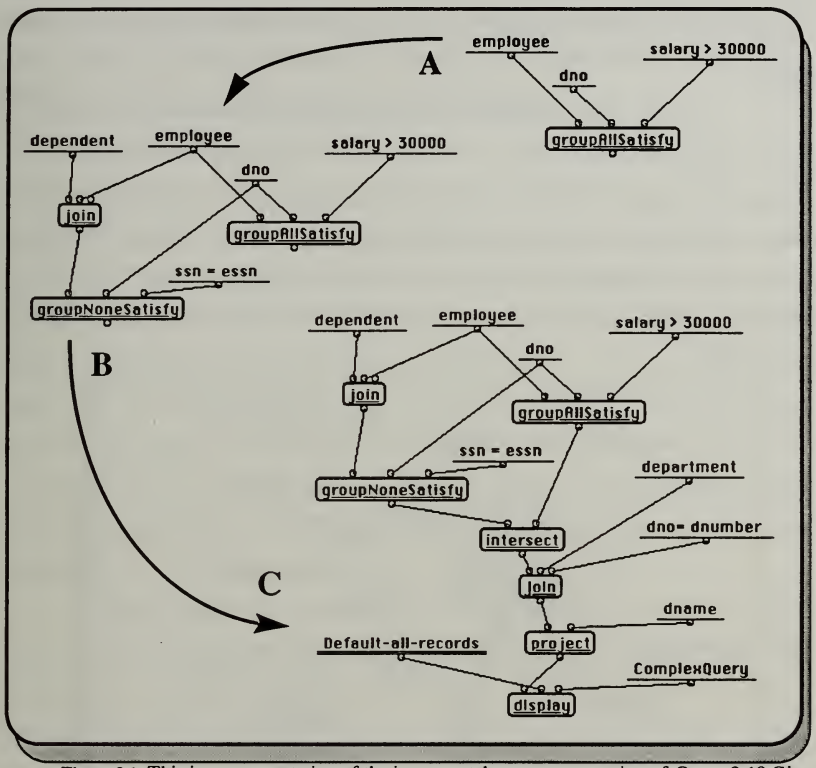


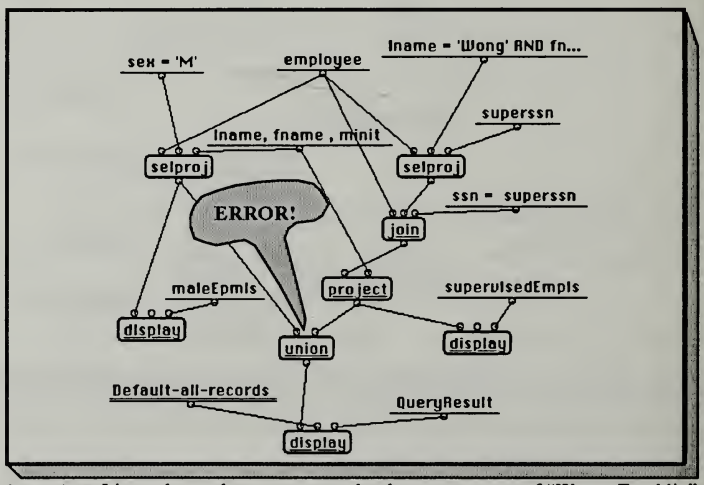
Figure 3.1 This is a representation of the incremental query construction of Query 3.19 Give the department names where all employees have a salary greater than $30,000 and have no dependents. showing the process level by level.

b. Incremental Execution

This feature is very helpful while debugging complex queries. If a complete query does not produce the desired results, it must be checked level by level to determine the erroneous part. The user should be allowed to see the intermediate result at any level by

executing the query incrementally. In our implementation, the user can double click at any operator's output node to execute the query up to that point and see the results. Also, the user can click in only those places to see the structure of the relation resulting from that operator.

This is a quick debugging method of complex queries, but the user usually cannot remember one intermediate result while investigating another. The user may sometimes want to see all of the intermediate results to make a comparison and determine the area to fix. In this case, he can use more than one display operator with appropriate alias names attached to desired points of the query and run the entire query to get the results for each display operator. Hence, these results can be checked simultaneously to give the exact idea of query.



Query 3.17 List male employee names under the management of "Wong, Franklin".

Query 3.17 is ready for debugging with intermediate display operators attached to desired points of the query to determine the errors. Above the final display operator, two additional display operators are attached to the output nodes of the *project and selectProject* operators. These display operators are given aliases of

"supervisedEmpls" and *"maleEmpls"* respectively. After the execution of the query, three results are provided to each display operator. Two intermediate results are correct according to the query; attention is then focused on the operator *union,* which must be an *intersect* operator in order to perform the AND conjunction. Therefore, the wrong operator is found easily by comparing all of the results at the same time.

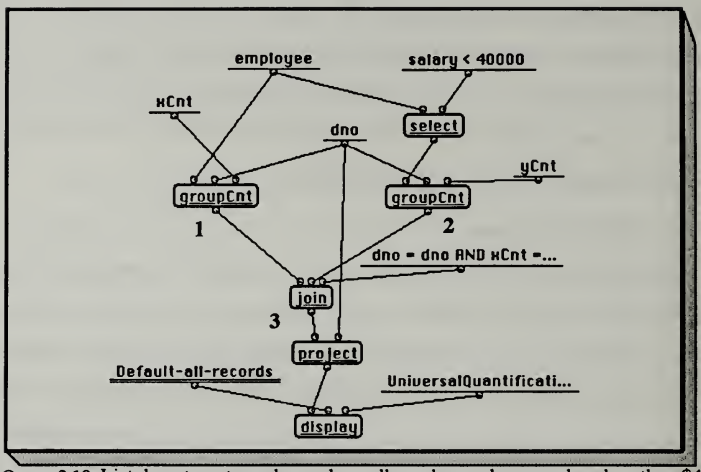
2. Universal Quantification

Expressing a universal quantification is very difficult in SQL as discussed previously. However, DFQL can use simple counting logic to achieve universal quantification. In other words, if all tuples in a relation or a group must satisfy the specified condition, we first count the numbers of tuples that meet the condition and then compare it with the total number of tuples under consideration. If these two numbers are equal, than the universal quantifier has been satisfied. We have used this idea to implement *groupAllsatisfy, groupNoneSatisfy,* and *groupNSatisfy* operators. The user can easily build his own quantifications as user-defined operators using the same concept, because this concept is easier to understand than the universal or existential quantifications.

In Query 3.18, the implementation of *groupAllSatisfy* is done by primitives to achieve universal quantification. The same counting concept is applied here: the number of employees is counted in each department at *point one*, and the number of employees which satisfies the condition specified in the query is counted at *point two*. The join operator is used at *point three* to get only the tuples which are the same in both relations. A project operator passes the department numbers to be printed to the screen.

3. Nesting and Functional Notation

The nesting feature in SQL exists naturally in DFQL. One by one execution of operators to supply input data to other operators similar to the execution of an SQL query from inside out, level by level. The lack of range variables and scoping rules in the nesting feature of DFQL improves readability and orthogonality.



Query 3.18 List department numbers where all employees have a salary less than $40,000.

Also, functional notation is used in all of the operators of DFQL to enhance orthogonality. Relational operational closure is implemented by the functional paradigm. Using operators that may take more than one input but produce only one output allows for easy combination into user-defined operators as previously discussed.

4. Graph Structure of DFQL Query

DFQL's visual representation of the query is a dataflow graph consisting of DFQL objects connected together by lines of dataflow paths. This representation adheres to the structure of relational algebra for the execution of the query. This graph structure provides two benefits:

- The internal operations of RDBMS's are based on relational algebra, therefore, relational algebra can provide a common interface to a DBMS without the need for a separate interpreter/compiler.
- DFQL can be optimized by a large number of techniques developed for the optimization of relational algebra expressions whereas most of the SQL interpreters/ compilers are not capable of performing optimization across levels of a nested query.

By using a graph structure of relational operators, the query can be more easily optimized than can combinations of partial queries in a textual block structured language. Actually, work already done [Dadashzadeh90] for converting the SQL queries into relational algebra graphs for optimization purposes result in structures quite similar to DFQL queries. By using a graphical, relational algebra approach to query formulation, the user is provided with a much more consistent and straightforward interface to the databases

C. PROS AND CONS OF DFQL

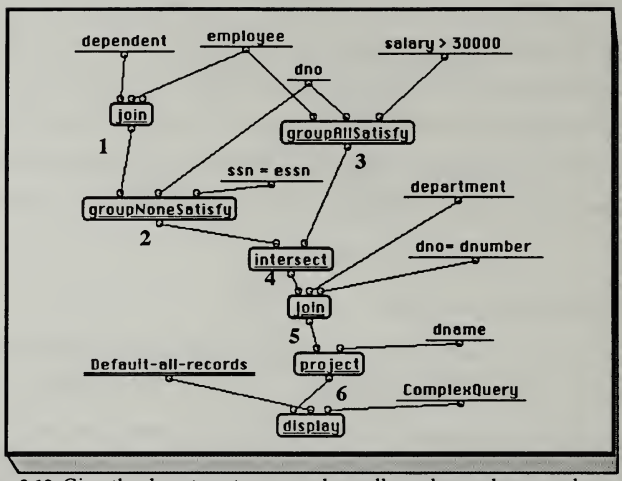
After discussing all of the features of DFQL, the advantages and shortcomings of this query language are presented. The merits of DFQL are related to the combination of visual representation, its dataflow structure, and its operator set. By these characteristics, DFQL provides the user the ability to easily express both simple and complex queries intuitively.

1. Power of DFQL

DFQL can express any kind of query very easily and efficiently using its powerful primitive operators. As mentioned previously, it is relationally complete; therefore, it has all of the relational operators including set, grouping, and aggregate operators. It can express universal or existential quantification by using only one primitive operator. To show the power of DFQL, an example query previously given in SQL (Query 2.3) is provided in Query 3.19.

DFQL has the necessary operators to formulate the query in Query 3.19 and will be explained in the next sections. The major role is played by two operators, *groupAllSatisfy* and *groupNoneSatisfy*. At *point one*(5), two relations, *dependent* and *employee* are joined together to get a cartesian product of all the possible tuples to be used by the next set of operators. The resulting relation of *dno* (department numbers for "all employees with no dependents") is determined at *point two* from the operator

(5) The numbers printed near the output nodes of operators in queries are not related to the queries themselves but are used to point to specific areas during the explanations.



Query 3.19 Give the department names where all employees have a salary greater than $30,000 and have no dependents.

groupNoneSatisfy, according the condition (*ssn* = *essn*) which retrieves tuples of employees with no dependents. For "departments with all employees having a salary greater than $30,000", we use the operator *groupAllSatisfy* with the mentioned condition and get the resulting relation *dno* at *point three.* Now, the results for both sets of conditions in the query have been satisfied, making these results union compatible. The *intersect* operator is then used to make an *AND conjunction* for the two conditions at *point four.* Department names are desired as a result instead of the department numbers which is why the relation *department* is joined at *point five.* This provides the names of the departments easily and passes them by a *project* operator to *point six.* A *display* operator is used to display the result of the query in a default form and title.

2. Extensibility

Extensibility is an important benefit of DFQL. The user may extend the query language by defining his own user-defined operators from a provided set of primitive operators, or the user's own previously defined user operators. User-defined operators can

be used as new user operators at any level of nesting and number. As used in Query 3.1, *selproj* is defined from two primitive operators to abstract the processes of *select* and *project* in one operator. These operators are constructed in a way that fully supports relational operational closure and makes them compatible with other operators. Once a user-defined operator is properly defined, it is completely orthogonal with the provided primitive operators.

By using user-defined operators, common operations for any given user can be provided at whatever level of abstraction is needed. For example, the user may like to see SSN, LNAME, MINIT, and FNAME as the result of his specified queries in his defined form. However he may not want to use the *project* and *display* operators repeatedly and put them in the same form. In this case, he can simply create a user-defined operator, namely *MyDisplay* and use it at the end of his queries to see the same information format. These extensions are entirely user dependent and each user can create his own style of working.

3. Ease-Of-Use

DFQL has the capability of representing complex problems intuitively with the aid of abstraction (embedding lower level details into user-defined operators), which is very useful, especially when combined with the visual feature of DFQL. This is because graphical representation is also easy to read and the concepts, once learned, are easy to remember. In the DFQL paradigm, relations are visualized as objects flowing from one operator to another. The ability to view relations as abstract entities directly contributes to the advantage of DFQL.

Since this language is orthogonal, it is both syntactically and semantically easier to use than other SQLs and it provides consistency, predictability and naturalness through the use of its operators. This feature is enforced in the user-defined operators as well, so that every user-defined operator must be operationally closed as well. Also, because of operational closure, the user is always certain of this result using the operators in this

language, which provides for greater ease in usability. These two features, orthogonality and closure support the user's ability to write error-free queries.

Another important ease-for-use feature of this language is the ability to create incremental queries. This is absolutely essential for the user to see the intermediate results of the partially built query and to continue building the query according to the intermediate results. Since DFQL is operationally closed, the user can feed each intermediate result to other operators, including user-defined operators. While building queries incrementally, the user can use temporary display operators to see the results. Also, the user can double click the roots of any operator to see the same kind of result within the default format. This provides the flexibility of changing incorrect queries at creation time.

4. Visual Interface

The various benefits of DFQL mentioned above are possible because of its visual interface, which is the basic advantage of dataflow programming and DFQL. Although building queries incrementally, grasping concepts easily, and encapsulating details in user-defined operators are advantages of DFQL, these are the merits of a visual interface too which do not exist in text based interfaces. This feature gives the user the ability to easily and interactively manipulate the DFQL query on the computer screen.

Having discussed the advantages of DFQL, the results of a human factors analysis of DFQL [Clark91] is used to compare DFQL and SQL. In this experiment, several students from different backgrounds and experiences are asked to develop three queries for each query language.Data is taken about the correctness, time of completion etc. After a few calculations, percentages of correctness are found. The results are presented in Table 3 to show the advantages of DFQL over SQL.

According to these results, DFQL has a higher percentage than SQL. In the technical and nontechnical category there is a difference of approximately 10% in both the DFQL and SQL percentages. In the experience category there is a difference of

	CATEGORY	NUMBER	% CORRECT	
			DFQL	SQL
1	Technical	18	50	30
	Non-Technical	8	42	21
2	Experience > 1 yr.	14	52	29
	Experience <= 1 yr.	12	41	25
	Total Sample	**26**	**47**	**27**

Table 3: HUMAN FACTORS ANALYSIS OF DFQL OVER SQL

approximately 10% in the DFQL scores and only 4% in the SQL percentages. While the 4% is not in itself statistically significant, a possible explanation for the discrepancy is that the technical background factor may be more important than the programming experience factor in the ability to use SQL. This implies that DFQL is easier to use than SQL for the people with a nontechnical background.

The shortcomings of the user interface design of DFQL and the problems related with the visual nature of DFQL itself are now discussed.

5. Interface Problems

The problems in this area are typical of problems seen in most visually oriented applications today. Typical screen size limits the number of the DFQL objects seen at once. As the complexity of the query grows, the objects in the drawing area become cluttered. This problem is temporarily solved by making the drawing area scrollable. When there are too many objects to be seen all at once, the user can scroll right/left or up/ down to a new drawing area. But this is still not a solution, since the user cannot see the entire query at once to comprehend its construction.

When many dataflows are connected to operators which intersect each other, the query becomes less readable and difficult to follow. This problem is also related with the size of the screen and drawing area. A solution to both of these problems is to utilize user-

defined operators to their fullest. In other words, when the screen becomes too cluttered, encapsulate some portion of it in a user-defined operator to make the drawing area more readable.

6. Language Problems

Within its bounds, DFQL is a very good query language. But when it comes to embedding this language in a textual computer language, some problems are encountered. Incorporating graphical data into a textual form while keeping the meaning and readability of the query intact is very difficult. DFQL queries can be compiled and inserted into textual programs as functions. However, this provides a poor way of looking at the DFQL code in the context of the program.

A solution to this problem is to translate the DFQL code to a textual representation keeping the same meaning of the DFQL code. But this is still a problem, since interpreting dataflow oriented languages such as DFQL into a purely procedural language is not easy. Since in the implementation of Amadeus, all back-end connections to RDBMSs are by means of the different dialects of SQL, another solution to this problem is to use the resultant optimized SQL translation of the DFQL query. This can solve the problem of incompatibility between textual and graphical representations. For this solution, however, the embedded SQL translation will not have the same advantages of DFQL.

Up to this point, DFQL has been discussed. The details of *Amadeus* and its implementation issues are discussed in the following chapters.

IV. FEATURES OF AMADEUS

Amadeus is developed as a prototype front end system capable of connecting to multiple back-end relational databases. It uses a graphical query language called *DFQL* for manipulation of databases. Communication is accomplished with each back-end using its individual dialect of SQL. Because of Amadeus' object-oriented implementation,(1) it is very easy to add new back-ends simply by adding the related classes of these back-ends called *database connectors.*

The design of this prototype incorporates modules which allow the user to build a complete application and provides efficient interoperability between that application and different connected RDBMS. Objectives of this prototype are:

- to provide easy to use, but powerful common language to access various types of RDBMS
- to shield the complexity of the underlying RDBMSs
- to allow a multi-user, and multi-back-end environment while enforcing the security measure for databases
- to provide error-free work for the user by implementing continuous error controls, warnings, and helpful information
- to extend the capability of this prototype to non-RDBMSs
- to allow the user to make conceptual, structural designs interactively and manipulate the resultant database without any conflict.

Some of these objectives are achieved in Amadeus, while the rest will be implemented in future research. According to the objectives, the user can design the database conceptually, translate it to a relational database, and manipulate it within the concept of the relational model. The relational model adheres to conceptual, structural designs and for manipulation. It is expected that users who know the relational model will very easily be able to understand the working style of this prototype.

(1) Implementation issues of Amadeus will be discussed in the next chapter.

Amadeus uses different modules to perform different tasks.Not all of the design modules are implemented, but Amadeus has the necessary modules to achieve the design objectives. The design modules of Amadeus are as follows:

- Query Editor,
- Relation Editor,
- Database Editor,
- Database Connectors,
- Interface Editor,
- Program Editor,
- Database Administration Module,
- Conceptual Design Module, and
- Network Connection Module.

Each module is discussed separately in the following sections. These sections also will be a guide to efficient use of this prototype.

A. GENERAL FEATURES

Amadeus is a complete program that can connect to back-end RDBMSs. These back-ends must be running prior to establishing connection with the prototype. Front end and back ends(2) are currently run on the same computer for now, since the *Network Connection Module* is not yet implemented. When Amadeus is run, it checks the user information file in the current directory. If the user information file does not exist, Amadeus warns the user to find the folder containing that information. Since this prototype has been implemented for one user, multi-user functions are not enforced.

Upon execution of Amadeus, three pull-down menus appear as illustrated in Figure 4.1. At this stage, only *new* and *open* sections are active which allow the user to create a new database or open a previously defined database. If the user wants to create a new database, a dialog box asks the name of the back-end in which this database will be included. The user makes his selections from the available back-end options provided in a scroll list. After the selection, a *database editor* pops up to define the new database

(2) We have implemented only one back-end connection, the *Oracle RDBMS*.

according to the selected back-end. When the user selects *open* from the database menu, Amadeus reads the database definition file for that user and asks the user to select the database desired to be opened. After opening or creating a database, the *schema* window, which contain a list of available tables belonging to the current database pops up automatically.

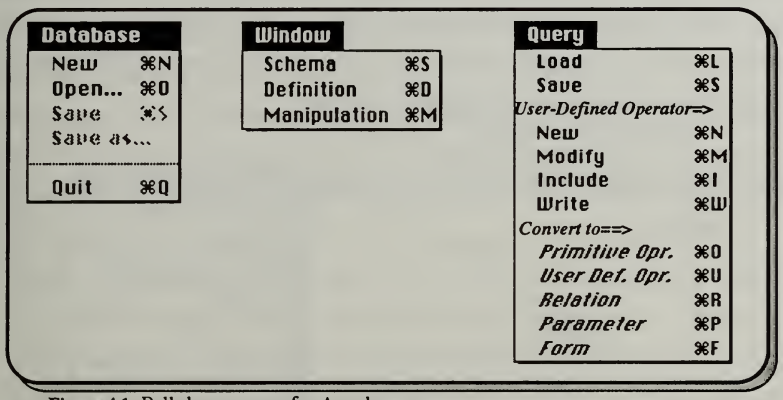


Figure 4.1 Pull-down menus for Amadeus.

The user can open the *definition* window to define a new table or the *manipulation* window to define queries to be used with tables from the *definition* window menu. The *query* menu is not active until the *manipulation* window opens, since it is only related with database manipulation. When the *query* menu becomes active, the user can *load* or *save* queries according to standard file operating procedures used in the Macintosh operating system. In the "*user-defined operators*" section of the *query* menu, the user can create or modify a user-defined operator (explained previously) and save it. He can also include in his system a user-defined operator defined by another user or in another database.

Once the user has finished his work, he can save the current database with the same name or with another name using the *save* or *save as* functions of the *database* pull-down menu, respectively. If the modifications have not been saved, when the user clicks the *quit*

option from the same menu, Amadeus asks whether or not to save the current database before quitting.

B. QUERY EDITOR

The query editor is for creating and executing DFQL queries. This window is designed to provide the previously discussed advantages of DFQL. As shown in Figure 4.2, the query editor has a query drawing area, several pop-up menus which are used to insert various query objects into the graph, and several function buttons.

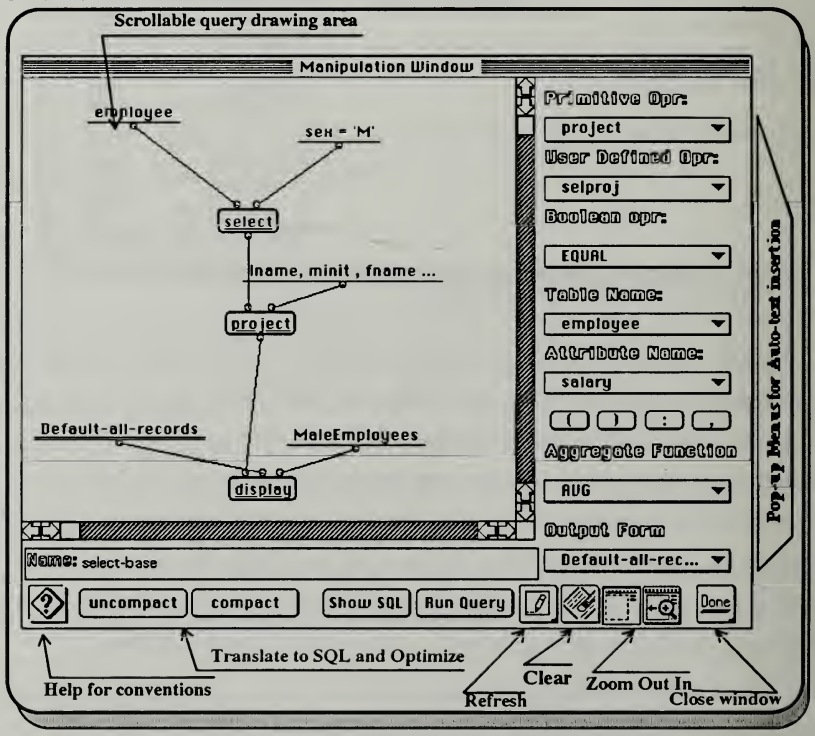


Figure 4.2 Manipulation window to define and execute queries in Amadeus.

Before explaining query construction, some key conventions to be used during the construction are explained in Table 4. These conventions are necessary in order to make the

process of query construction easy and error-free for the user. There are seven pop-up menus containing necessary information to be inserted automatically during construction of the query. These pop-up menus are dynamic in that they contain different items, based on the current database, user information, and the connected RDBMS. The contents of the pop-up menus are called *table name* and *attribute name*, depending on the current database. The *attribute name* menu contains only the attributes of the table currently in the *table name* menu. Each time the current table is changed, the related attributes are loaded into the *attribute name* menu. The pop-up menus *user-defined opr.* and *output form* contain the user-defined operator and form object names available for the user. The contents of these menus change according to the definition of the new forms and user-defined operators. The rest of the menus are related to the connected database so that only the allowed aggregate functions and boolean operators appear in them. This feature is extremely helpful preventing the user from using misspelled names or using attribute names which are not defined. This also provides some convenience to the user during the creation of queries. The user does not have to worry about whether a function is allowed by the connected RDBMS, nor does the user have to memorize attribute names present in each relation of the current database.

Four buttons are provided for the purpose of inserting frequently used characters during query formulation. By including these buttons, the user can construct his query without typing from the keyboard. This also reduces the possibility of errors in the query from typographical mistakes. There is a scrollable query drawing canvas which is sufficient in size for an ordinary query. A *name section* has been included under the drawing area to show the name of the current query. This is very useful when zooming in to or out of user-defined operators in the query. This feature updates the exact path, similar to a conventional directory description update.

Some operational buttons are included at the bottom of the window. Some of them are icons that perform standard operations like closing the window, zoom out or in, clear, refresh, and help. Zoom in and out buttons are used to traverse into the user-defined

REGION	OPERATION	ACTION
DFQL Object's body	To select the object.	Click on body.
	To select multiple objects.	Click on each body one by one.
	To move an object.	Click on body, move mouse while pressing button, release when done.
	To move multiple objects.	Select (highlight) each of them and perform move as described above.
	To see the operator description, change the text of an object, or launch the interface editor.	Double click on an operator, on an object, or on a form object respectively.
	To change object type.	Select object and use the *query* menu to convert it.
	To see the contents of a user-defined operator.	Select a user-defined operator and click the *zoom in* button (with the plus sign).
Root (output node) of a DFQL object.	To start or finish drawing a line connecting a terminal.	Click on it to finish or click on it to select (highlight) and drag the mouse with the line.
	To see the table structure of the resulting relation.	Double click on the root of the operators only.
	To see the result of the query as the resulting relation of the operator.	Hold the command key and double click on the root of the operator for partial execution.
Object's terminal (input node).	To start or finish drawing a line connecting with a.	Click on it to finish or click on it to select (highlight) and drag the mouse with the line.
	You cannot double click on any terminal for debugging purposes.	
Space drawing area.	To deselect the selected objects.	Click on area where there is no object's body or nodes.
	To create a DFQL object.	Hold the command key and click on an area where the object will be drawn.
DFQL Object.	To insert operator, table names, or functions in objects	Hold the command key and select the desired entry from the pop-up menus and release it after the text is inserted into the current dummy object.
	To delete a DFQL object	Select (highlight) the object to be deleted and hit the <BackSpace> or <Delete> key from the keyboard.

Table 4: KEY CONVENTIONS FOR QUERY CONSTRUCTION

operators from the query to see the formulation of these operators. The user can go back and forth as long as there are user-defined operators defined to investigate. This feature is very useful in understanding the exact process of these operators, since some of the user-

defined operators may not have been formulated by the user himself and have been included in other users' databases. The user can refresh, clear, or get information about the key conventions any time by clicking on the individual buttons.

The user can execute the query by pressing the "*Run Query*" button. In order to do that, since this is not a partial execution, the query has to contain at least one display operator. The button *"Show SQL"* can be used to see the optimized SQL statement used to perform this query in the back-end. This is possible because each communication can be performed by SQL. This is available only when the query is finished and *compacted*. Once the query is compact, it can be seen but it cannot be modified. In order to modify the query, it must be *uncompacted* first. To aid the user, compacted queries have a differently shaded background to indicate that no alteration is allowed.

1. Construction of Queries

After opening the *manipulation window*, the query editor is ready to formulate a new query. The user also has the option of loading a previously defined query to modify or execute. A DFQL object can be created by holding the *command key* and clicking at the exact point in the drawing area where the object will appear. A dummy rectangle appears on the screen with a text cursor inside to type or insert the name of the object. The user can either type the name or insert it automatically from the pop-up menus. The name of the operator or text is pasted into that area. This process is continued sequentially to formulate an attribute list or condition list as long as needed. To draw the actual operator or object, the *return key* <*CR*> must be pressed. The query editor then identifies the written text and draws the matching DFQL object in the same spot, after clearing that portion of the screen. By using automatic insertion, it is assured that every object is defined in the current database. Objects can be connected by clicking on a terminal (root) of the desired object, and then drag the moving line to the desired root (terminal) of the other object and clicking again. The connection is established if the click point is a valid terminal or root. The user cannot connect a root (terminal) to another root (terminal), because DFQL

requires that the data must flow from the output node (root) of an object to the input node (terminal) of another object. If this rule is violated, a warning message pops up as seen in Figure 4.3. Root objects can be connected to more than one terminal, allowing the use of a single result in several places, but this is not valid for terminals since only one data flow is allowed into each terminal. If a new connection is made to a terminal object, the old connection is deleted. This provides the ability to change connections very easily. Repeating the connection process on a previously connected line deletes that line.

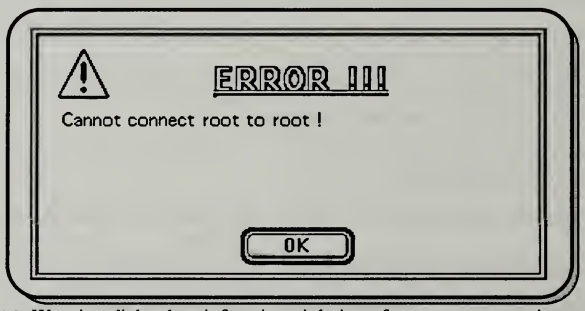


Figure 4.3 Warning dialog box informing violation of a query construction rule.

a. Complete Query Construction

The entire query construction is finished when the required DFQL objects are drawn and their connections are complete. As previously mentioned, in order to finish the query formulation, at least one *display* operator must be present in the query. This is because every query result has to be printed on the screen in the given format and with the given alias name. After the query formulation, the user can run the query to check its correctness. A complete error checking is done during this execution, since some terminals may not to be connected or they might be connected to disallowed objects. As an example, a *select* operator has two input terminals in which one terminal input is a relation and the other terminal input is a condition, but the user may accidentally reverse the order, or may have forgotten to connect one of terminals. In this case, a warning dialog box pops up explaining the exact error and the related object blinks in the diagram to indicate the area

requiring correction. The diagram can be reset simply by clicking on an open area to stop the blinking. This kind of error checking cannot find semantic errors when the query works properly but produces results different than desired.

b. Incremental Query Construction

Incremental query construction is another way of formulating the query is explained in Chapter III. The user can divide the query into logical sections to formulate part by part and then combine each part. In each section, the user has to be sure the result is correct. To do this, the user has to see the relation structure or the values produced. If the result is incorrect, the user must correct that part and then continue query formulation. It is easy to fix a small part of the query rather than the entire query. Two features are provided in the editor allowing the user to see the table structure and the values of the tables at any point of the query. The user simply clicks or (double clicks) the root of the operator to see the table structure or (values of the.table) created as an output from that operator. After checking the result, new sections of the query can be built on the existing sections with the knowledge that the query is correct so far.

2. Formulation of User-defined Operators

To take advantage of the merits of user-defined operators in DFQL, the capability to define user operators is included in Amadeus. The user can initiate formulation of user-defined operators by selecting *new* from the *query menu*. The user is asked to specify the number of input terminals for the user-defined operator, and then two shaded bars (top and bottom of the window) are drawn with the specified number of terminals as shown in Figure 4.4. The user can then formulate his user-defined operator by connecting the terminals and the root of the in and out bars. Two circumstances exist which will prevent the root of the output bar from connecting to a terminal: when a *display* operator exists in the current user-defined operator or there is another user-defined operator containing a *display* operator.

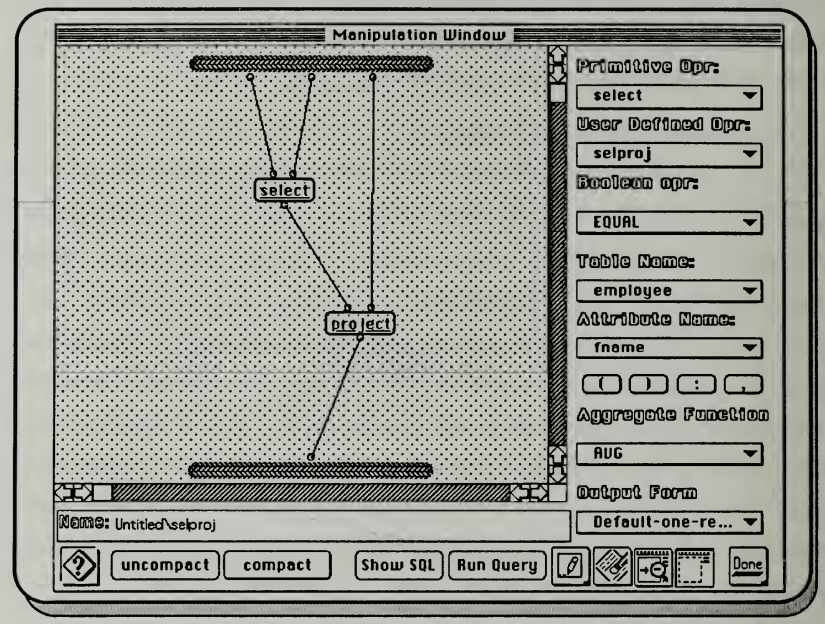


Figure 4.4 Creation of a user-defined operator in the manipulation window.

The number of input terminals can later be changed by simply double clicking on the input bar and selecting the new number of terminals from a dialog box. After the formulation is completed, it must be saved by selecting a *write* command from the *query* menu. Before the formulation is saved, however, a name and an explanation of the input/ output connections are requested by a dialog box. This process provides the user more information when he double clicks on the defined operator. The newly defined operator is then included in the system and the *User-Defined Operator* pop up menu is updated making it available for use in future query formulation. The user can then retrieve the definition of these operators at any time to make modifications. He can also include any other user-defined operators in the system to be used with the current database.

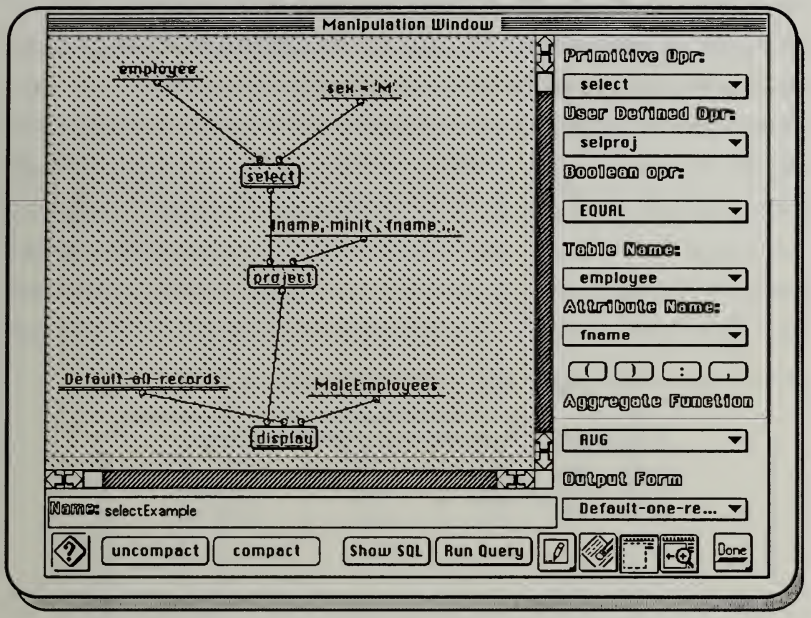


Figure 4.5 Compacted query shaded with a pattern to indicate that no modification is allowed other than traversing into the user-defined operators.

3. Query Execution and Debugging

The user can execute the query in two ways. The first is complete execution of the query by clicking the "*Run Query*" button. In order to do a complete execution, there must be at least one *display* operator or a user-defined operator including this operator. The result will be displayed according to the inputs of that *display* operator. The second query execution method is to execute query up to a certain point by double clicking at the root of an operator or relation object. The result will be displayed in a default form titled "*DISPLAY*". This method is especially suitable for debugging purposes when partial results are need for investigating.The user also has the capability of using more than one *display* operators to see intermediate results while executing the entire query.

There are two types of queries in the Amadeus prototype: one is an the actual defined query and the other is a compact query with an optimized SQL translation. A compact query can only be executed by its optimized SQL translation through the back-end and cannot be modified. Since any modification to a compact query can change the translation of the SQL query, it must first be converted to an uncompact type before any alteration is performed. By keeping the queries in compact form after definition, they may be used in application levels without fear of modification. A button is provided to see the optimized SQL translation, as illustrated in Figure 4.6. This button is inactive if the query is not compact, since it does not have an SQL translation(3). If the user tries to modify any portion of a compact query, a dialog box pops up to warn the user.

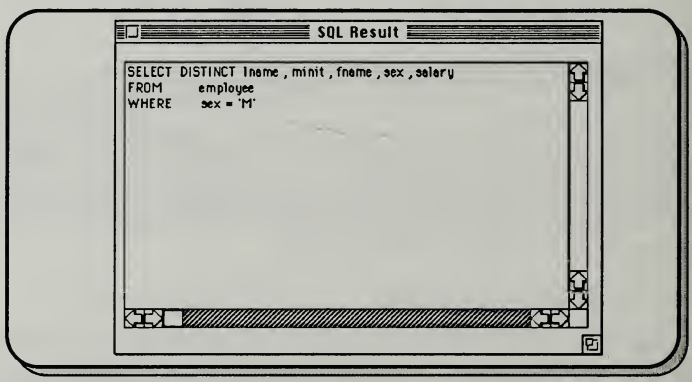


Figure 4.6 The SQL result of a compact query that is used between back-end and Amadeus.

4. Display of Query Results

As previously explained, only one *display* operator is available to print the query results to the screen. To display the values of the tuples of a relation, two extra inputs must be defined, namely the form object and an alias name. The alias name is printed as a title

(3) The difference of compact and uncompact queries are discussed in detail in the next chapter.

in the result window to distinguish multiple result windows. This prevent confusion when using more than one *display* operator in a query for debugging purposes.

Two default forms exist in every application available to the user, selected from the "*Output Form*" pop up menu. A user can also define customized forms by using the *Interface Editor* (explained in Chapter V). This editor can be launched from the *Manipulation* window by double clicking on the form object. The first output form is provided to display all the tuples of a table at once. Each tuple is displayed in one row, as seen in Figure 4.7. The form window is resizable, and depending on the number of columns of the resulting relation, it can be adjusted to see the whole table at once. It is also

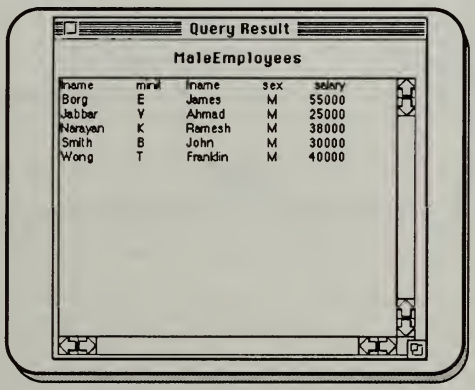


Figure 4.7 The default output form to display all tuples of a table at once.

scrollable in each direction as well, making possible to keep the window small and see the other portions of the table as well. Attribute names are printed at the top of each column. The values of each column are aligned according to the type of values being displayed. For example, numeric, string values, and characters are justified right, left, and center, respectively. Additionally, the user has the option of changing the alignment of the columns by clicking once on the column area. The width of the columns can be changed by clicking the edge of the column and dragging it left or right. These features provide ease in investigating the results provided in this default form.

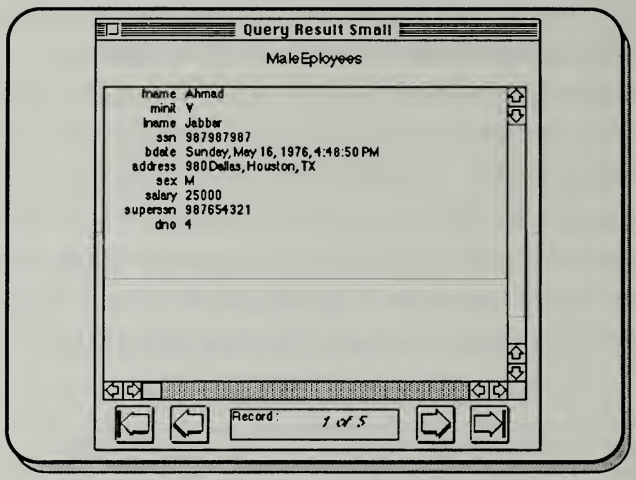


Figure 4.8 The default output form to display one tuple of a table at a time.

The second default form allows the user to see the results one tuple at a time, as seen in Figure 4.8. This form can traverse the table record by using the buttons at the bottom of the window. The user can also go to the top or bottom record of the table using two other buttons, also at the bottom of the window. To indicate place in the search, a record number is printed, giving the total number of records as well. The attribute names are printed at the beginning of each value. These are right justified to make the form more readable. Features of changing the justification and size of the column are permitted as well. In both of the output forms, a title is printed using the alias provided to the *display* operator.

5. Help features

Help features are included in the *Query Editor*. One feature gives a description of the query operators, including the user-defined operators. This description includes the names and input objects that are supposed to be connected to the terminals. An explanation dialog box pops up when the body of an operator is clicked, as illustrated in

Figure 4.9. This gives the names of the input objects and their order from left to right. Referring to the example in the figure, the *select* operator takes two inputs, relation and condition, in that order. It produces a relation from the input relation by selecting only the tuples which match the condition. This feature allows the user to view all the operator connections and their exact orders, obviating the need to memorize them. This is convenient when user-defined operators are used in the query, since of the user operators could be imported from other users' definitions.

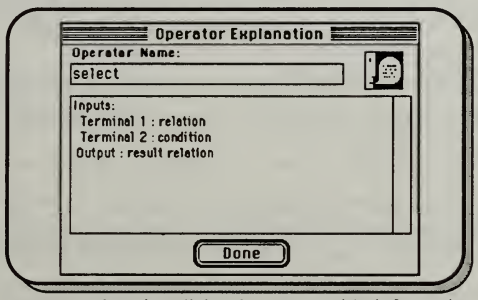


Figure 4.9 Operator explanation dialog box to provide information about each DFQL operator.

A complete on-line help window is provided for the key conventions used in this editor, as described in Table 4. The user can open this window by clicking on the "*Help*" icon at the bottom of the *Manipulation Window*. This help window has a scroll list of operations; once an item is clicked in the scroll list, the related explanation appears in other multi-line text item. This very simple but useful help window provides continues on-line help and is shown in Figure 4.10.

C. RELATION EDITOR

This editor provides a window the user can use to define new relations or modify existing relations. As mentioned before, these editors are dynamically changed according to the connected back-end. For example, the user can define a type of attribute supported only by the connected RDBMS. As a result, there are differences in the created tables which

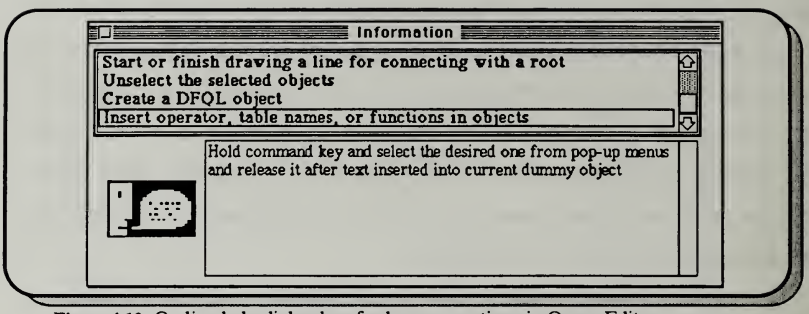


Figure 4.10 On line help dialog box for key conventions in Query Editor.

are not compatible with other back-ends. In order to use these relations in other RDBMSs, they must be converted and made compatible with the desired back-end.

As shown in the table definition window in Figure 4.11, the user can give the name of the relation and the attribute specifications used in the relation. The type of attribute can only be selected from the "*Type*" pop up menu which contains the supported attribute type of the currently connected back-end. Size and properties of the attribute must be specified in addition to the name and type of attribute. For the relational model, every relation must have at least one key attribute without duplicate values. These key attributes can be indicated as the properties of the relation. Besides the key specification, the user can specify whether the attribute can have a null value or not. After these definitions are made, the user can add this attribute to the table. He can also select an attribute from the attributes list to modify, delete, or change its place in the table. When the "*Create Table*" button is clicked the relation is created and included in the current database. This is a simple, but efficient editor to create and modify the tables of the relational model. This editor can be launched from the *"Database Editor"* (discussed next).

D. DATABASE EDITOR

This editor can be opened from the "*Window*" pull down menu to modify the current database, or from the "*Database*" pull down menu by selecting "*New*" to create a new

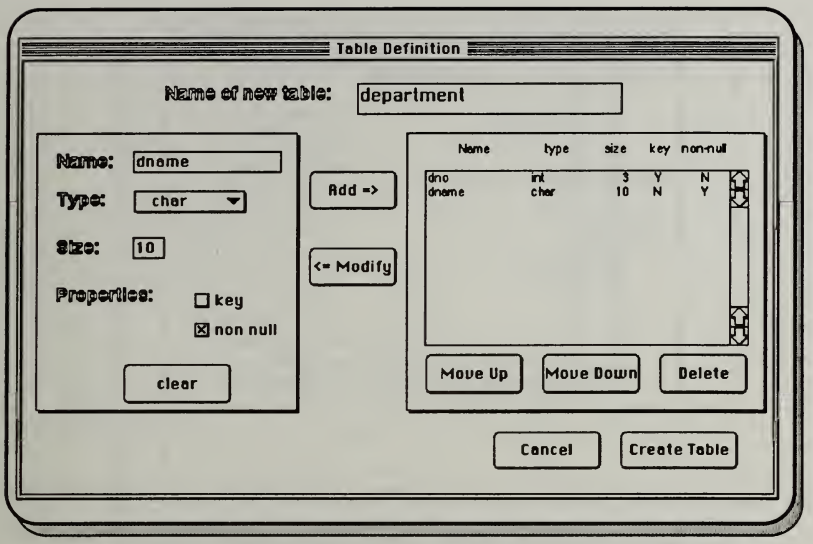


Figure 4.11 The table definition window allows the user to define or modify relations.

database. This editor allows the user to launch the table editor and then work with the two editors together. After definition of the tables, each table's name appears in the scroll list of this editor's window, as seen in Figure 4.12. The connected DBMS's name is provided to inform the user about the back-end. The user has the option of deleting, creating, or modifying the database by selecting the specific relation. To create the defined database, the "Create DB" button must be pressed. The definitions of the current database can be updated this way and saved with the same name or with a different name.Each database is specific to the connected back-end and database or relation definition cannot be done without connecting to a back-end. This is enforced by Amadeus to make sure no incompatibility occurs during manipulation.

E. DATABASE CONNECTOR

A specific *Database Connector* for each connected back-end is incorporated in Amadeus which encapsulates all the information and methods to communicate with each

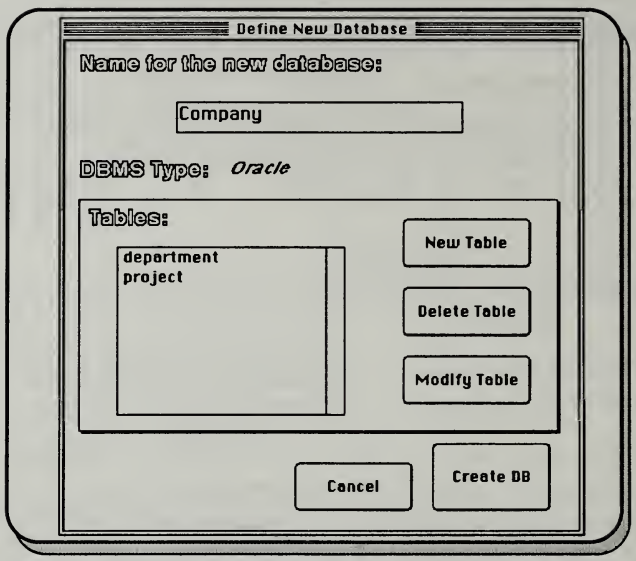


Figure 4.12 The database definition window allows the user to define or modify the databases.

back-end RDBMSs. Amadeus communicates using SQL statements particular to each back-end instead of using a kernel database connector to make the translation between them. Therefore, these connectors are responsible for connecting and communicating with the specific RDBMSs in their own dialect of SQL. This module must enforce the compatibility of the definitions and manipulations with the connected RDBMS.

After opening the desired database and establishing the connection with the back-end, the *Database Connector* opens a schema window that shows the available relations, as illustrated in Figure 4.13. The user can double-click on any table name in the window to open a table structure window showing the definitions of the attributes of the selected relation. The user can open as many structure windows as necessary to see the entire database relation definitions. The relation structure window shown in Figure 4.14 does not

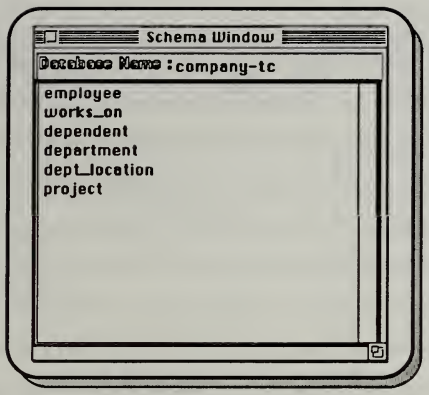


Figure 4.13 The schema window that shows the table names of the current database.

allow the user to make any modification. This feature is included here for information purposes only. Related implementation issues are discussed in the next chapter.

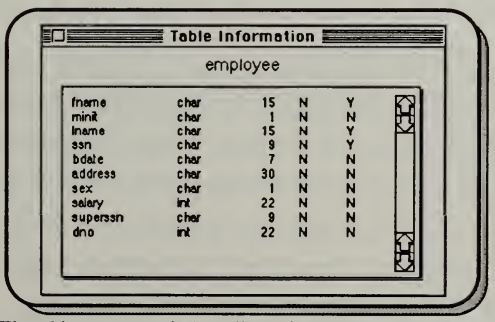


Figure 4.14 The table structure window allows the user to see the definitions of the attributes in relations.

F. INTERFACE EDITOR

This module is implemented for the same environment as part of another research effort [Hargrove93] but is not yet included in this prototype. This module can be invoked from the "*Window*" pull down menu or from inside the *Query Editor*. The interface editor is capable of designing customized forms in which the query result will be displayed. Each form has an associated DFQL form object and can be used in the queries after the definition of the form. Also, these forms are specific to each database and connected back-end like the other editors. These customized forms are called *output forms* and in the original implementation of Amadeus, are used only to see the data. This implementation of editor supports *input forms* that can be used to enter values in the tables and send them to the related *database connector* to do the update operations. Additionally, this editor allows the user to print the forms directly to a printer instead of the screen using the same format. The user can personalize his application by customizing his forms and can get hard copies of the results.

G. PROGRAM EDITOR

The *Program Editor* is not yet implemented in Amadeus. A complete program editor is needed to use definitions of a language that can create applications. A third-generation language capable of embedding objects created by other modules of Amadeus is needed to take advantage of the features of a third generation language. The problem for DFQL and Amadeus is determining a third generation language capable of holding these objects. The problems of structured sequential programming languages discussed previously prevent their use in a prototype that uses a visual graphical representation of queries. Therefore, a visual dataflow programming language is more suitable for Amadeus. A language like *Prograph* [TGS88a][TGS88b][TGS91], providing the advantages of visual and object-oriented programming, generated the design of the main features of DFQL. The necessary extensions can be provided to this language to make possible the use of every kind of object

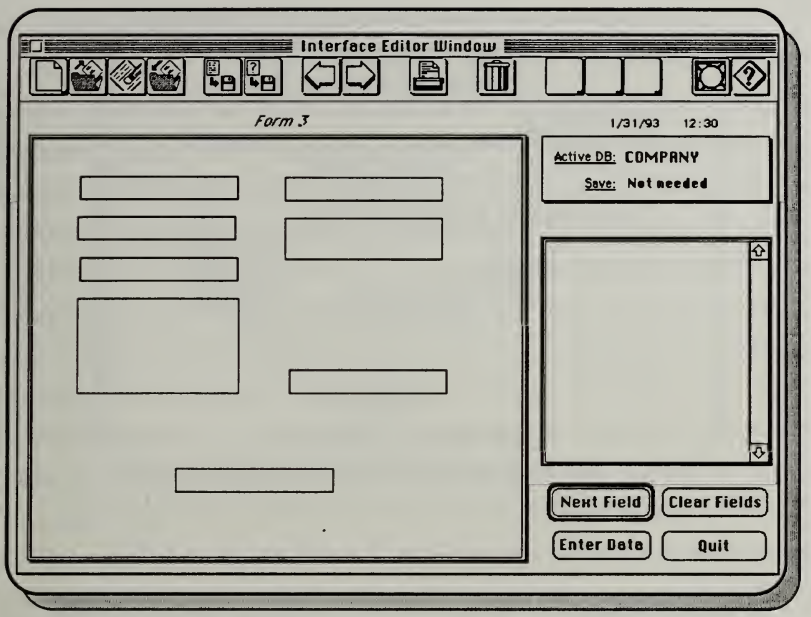


Figure 4.15 Interface Editor's window allows the user to define customized forms. [Hargrove93]

in this prototype. Since Prograph is a visual dataflow programming language that uses the object-oriented features, incorporating it as a program editor presents few difficulties.

Although it is not implemented, an example of incorporating Amadeus into *Prograph* as illustrated in Figure 4.16 (see Appendix B for language's syntax). In this example, a small method which can be considered as macros gets a list of query objects and executes a loop to run each query. This is done by selecting the related table and saving the list of results to the disk for future use.

H. DATABASE ADMINISTRATION MODULE

This module is responsible for all of the security issues of the prototype. This module is a very important component of Amadeus which must be implemented as a separate

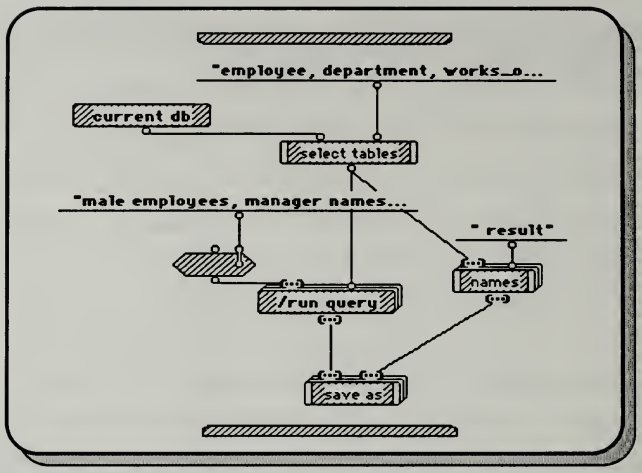


Figure 4.16 The definition of a macro for this prototype incorporated with *Prograph*.

research. For now, only a fixed, hard coded user name and password is available to access the back-end; all other security issues are open. This module can be used only by the *Database Administrator (DBA)* or a super-user who has all the responsibility of the security issues of the applications. Objectives of this module are as follows:

- A specific user name and password can be assigned to each user to limit the access to the applications and related stored data.
- To be able to specify access rights to owner, groups, and other users for the user created databases, forms, queries, and user-defined operators.
- To prevent the extraction of data from shared databases according to access privileges by enforcing some type of security model.
- To prevent conflicts of resources when multiple users try to modify them at the same time.
- To furnish sufficient back up procedures to protect the data and the applications from unexpected problems.

Once these goals are achieved, this prototype will be much more secure, and will add additional merit to this development.

I. CONCEPTUAL DESIGN MODULE

In the design of this prototype, a complete conceptual design module has been introduced expressing real life in a conceptual model. The prototype can convert the conceptual design to the relational model and create the necessary relations automatically. This feature is very useful, since from beginning to the end, the user has every kind of tool available to convert the desired features into an application.

This module is also not implemented and remains in the design phase level of the prototype. Amadeus is designed to use the ER-model [Chen76] as mentioned in Chapter II (Entity-Relationship Model Interface on page 11). In order to do this model, a graphical editor, like the DFQL editor, must be implemented to draw the ER diagrams easily. Then, an interactive translator must be implemented to convert this diagram into a relational schema. This must be interactive, since the relational model cannot represent all of the constraints expressed in the ER model. Therefore, the decisions of the user must be carefully considered during the translation to eliminate constraints that cannot be enforced in the relational schema.

J. NETWORK CONNECTION MODULE

This module is designed to use a network to connect the various back-end RDBMSs located in different places. This module is also not implemented because of hardware problems inherent in connecting Macintosh computers through a network. Including this feature in the prototype will prevent need to run the back-ends on the same computer. A local talk connection is established between computers but its data transfer rate is not fast enough for the prototype. Additionally, current memory capacity of Macintosh computers on hand does not permit running Amadeus and more than one RDBMS simultaneously.

Connecting the prototype through a network provides the flexibility of locating the back-end RDBMS virtually anywhere and solves the memory shortage problems of computers. This module allows the prototype to accommodate a large application that

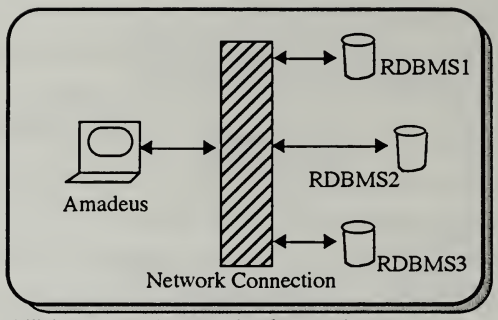


Figure 4.17 Utilizing a network connection for Amadeus.

works with many back-ends containing large databases. This module can be used as depicted in Figure 4.17.

In the next chapter, the implementation issues of Amadeus are discussed, including a discussion of object-oriented design and the application of object-oriented features in this prototype.

V. IMPLEMENTATION DETAILS

Amadeus is implemented using a a visual object-oriented programming language named Prograph (see Appendix B) which uses the data flow paradigm as an interface. This language is currently available in the Macintosh environment. Prograph was chosen for several reasons. First of all, its visual data flow structure is very similar to the approach taken for DFQL. This similarity helped stimulate the development of DFQL. Also, the ability of Prograph to take advantage of the Macintosh visual interface greatly aided in the development of the Amadeus prototype. Since Prograph is object oriented, it allows use of the many powerful features of the object-oriented paradigm. This also greatly improved the modularity and maintainability of the resultant code.

> Prograph is a "very high-level, pictorial object-oriented programming environment" that integrates four key trends in computer science: a visual programming language, object orientation, data flow, and an application-building toolkit. ([Wu91b] on page 77)

The *Oracle* relational DBMS, running on Macintosh computer with operating system version 7.0, is the only back-end currently connected to this prototype. Additionally included is the native database connector of a programming language, although its connection and other features are not yet fully tested. Both Amadeus and Oracle run on one computer, since a network connection module is not yet implement. The host computer's current memory capacity (8 MB) can run only the prototype and one back-end(1). For this reason, implementation of more database connectors for other RDBMSs is not done. This is not considered a major problem for this prototype, since it is very easy to add a new database connector using the object oriented features of Prograph.

(1) The use of virtual memory is limited, because of the degradation of data retrieval efficiency.

A. OBJECT ORIENTED DESIGN

This prototype is designed to take advantage of the object-oriented paradigm. These advantages are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Reliability
- Extensibility

These features make this prototype reusable, sharable, integrable, and extensible. Examples of using these advantages will be provided throughout the discussion of Amadeus' implementation.

Prograph has an application editor that allows the user to create menus, windows, and dialog boxes for an application.It also provides to the user all necessary classes for the application and the user interface. As depicted in Figure 5.1, the pull-down menus and windows are inherited from related system classes and the necessary methods and attributes are added to them. The *inheritance* feature of the object-oriented paradigm is used here to abstract the common methods and attributes in the parent classes whereas different methods and attributes are included only in needed child classes. As many instances as of the classes necessary to be used in our application can be created.

Since communication between classes is done by sending messages back and forth, a message can be sent by including the instance of a child class. If that child class does not have that method to receive the message, it propagates the message to the parent class. The common methods in parent classes are called very easily using this feature. This working style of object oriented languages enforce the reliability of the programs. There is no doubt that included classes can work together when they are integrated in an application. Since the individual modules are robust and error-free, they can be integrated very easily with a reliable mechanism of object oriented programs (OOP).

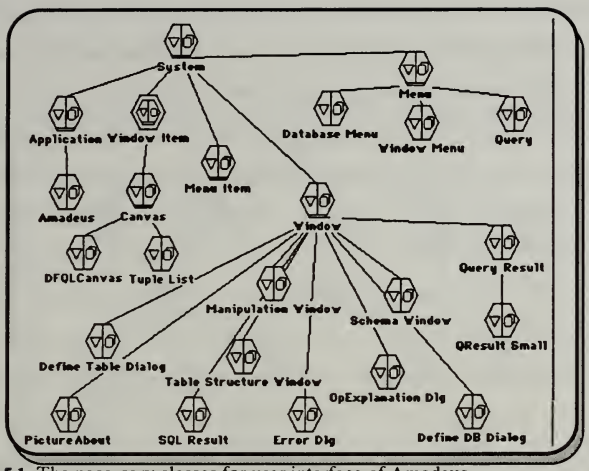


Figure 5.1 The necessary classes for user interface of Amadeus.

The DFQL queries can be saved on a disk and loaded back with the same graphical representations. Instances of relations and definitions of user-defined operators are not included in the storage file. These types of objects and definitions are linked together after retrieval of the query from the disk. The main reason for this operation is to keep this data updated and avoid using older versions of tables and user-defined operators in case of recent modifications to them. Updating is an automatic process right after loading the query. If a table no longer exists in the current database, a warning message appears and cancels the loading operation. Implementation of three main sections *graphical editor, SQL translation,* and *back-end connection* are discussed in the next sections.

B. IMPLEMENTATION OF GRAPHICAL QUERY EDITOR

As illustrated in Figure 5.2, two separate classes, *DFQLObject* and *Connector*, are used and the necessary child classes are inherited from them. The instances of class *connector are* used as attributes in the *DFQLObject* class and the instances of this class are stored in the class *DFQLCanvas* which is inherited from a graphical drawing window item called *Canvas*. The item *Canvas* is used in *Manipulation Window* as a main drawing area.

The class *DFQLCanvas* is responsible for controlling the drawing process for the *query editor*, but the *DFQLObject* and its child classes are responsible for storing all the connections, positions, sizes, and information related with the query.

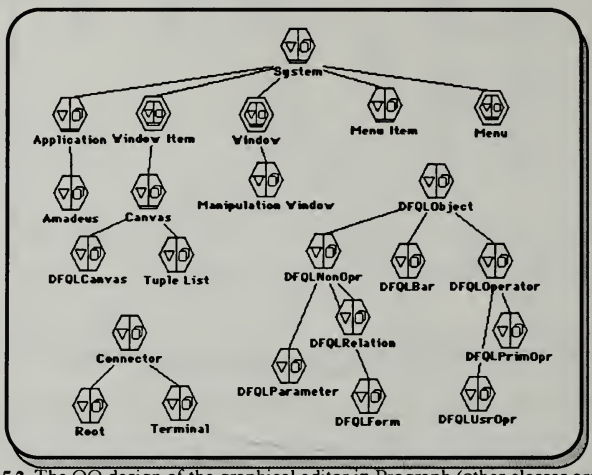


Figure 5.2 The OO design of the graphical editor in Prograph (other classes are not shown).

Three main sections of drawing objects that appear on the canvas, *DFQLOperator*, *DFQLNonOpr*, and *DFQLBar* are created as illustrated in Figure 5.2. The last section is used during the definition of user-defined operators. Two subclasses of *DFQLOperator* are similar to drawing process, but they have different contents. Primitive operators execute the main query operators whereas user-defined operators have a link to the contents of defined operators and establish the connection to its constituents during execution. *DFQLNonOperators* have a different drawing representation from the operators, and so have their own drawing methods. Non-operators such as *relation*, *parameter*, and *form* are separated from the operators. *Relation* stores the relational table in its attribute called *rootValue*, whereas *form* stores an object defined by the *interface editor*, and *parameter* has only text of the condition or attribute list in it.

The class *DFQLCanvas* consists of 13 methods that perform the control of drawing area as shown in Figure 5.3. The main role is played by the method called "process *click*" which gets the mouse clicks on the drawing area, finds the object at the click point, and then dispatches the related operation. It checks the click point to determine whether a terminal, root, or object's body has been selected. If the click is a *double click*, then the process click calls the related operation to start a partial execution, operator information, or text edit dialog box respectively. If the click is a *drag*, it moves the selected (highlighted) objects to the next point according to their relative positions. If a terminal or root is selected, then process click goes through a line drawing process. If none of the above explained happens, then process click deselects an object or creates an object if the *command key* is pressed.

The connection is determined by storing the terminal (root) object inside the other connected root (terminal) object. For example, to find the connected objects of an operator's terminals, simply get the list of the terminal objects stored in that operator, retrieve the values stored in the *connectedTo* attributes which are the root objects of the connected objects (it is NULL, if not connected). Then, the values of the attributes called *partOf* are retrieved which are the actual connected objects. This linkage is two way, so query execution may be traversed either way. All connectors know their places relative to position of their objects' bodies. Each time the body moves, the relative positions must be recalculated.

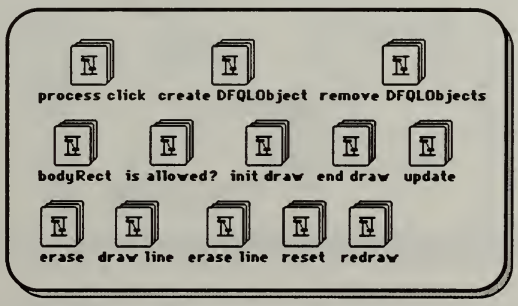


Figure 5.3 The methods of DFQLCanvas used to control the query editor.

The *draw* method used in the graphical editor provides a good example of *polymorphism*. This feature of OOP languages gives a big advantage to programmers by executing the draw method by giving the instances of different DFQLObjects. *Polymorphism* can then determine the related draw method, depending on the instance being used. An example of this convention is shown in Figure 5.4. This is usually used for refresh purposes and called by the system to redraw the canvas, or by the *process click* method to draw the objects again when a relocation or deletion occurred. In the figure, this method gets the DFQLCanvas as an input and retrieves the *DFQLObjects* stored in the same named attribute. This is a list of DFQL objects passed into the loop calling their related *draw* methods. Draw methods for classes *DFQLOperator, DFQLPrimOpr, DFQLNonOperator, DFQLBar, and DFQLForm.* are defined. There is no definition of draw method in the other object classes listed above since they have the same kind of DFQL representation. Their draw method is generalized into the class *DFQLNonOperator.* Also, *a draw* method defined in the class *DFQLPrimOpr* overrides its definition in its parent class, because the name of the primitive operators is written as underlined text. Therefore, a separate method to draw each individual object is not defined.

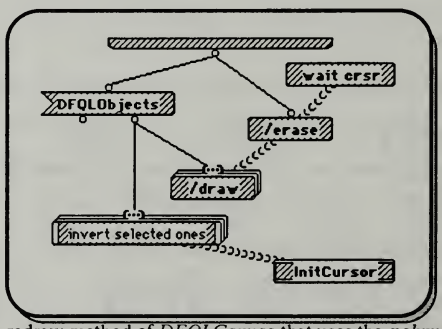


Figure 5.4 The redraw method of *DFQLCanvas* that uses the *polymorphism* of OOPL.

C. BACK-END CONNECTION

There are three classes created for databases namely, *Database, Relation,* and *Attribute,* that contain the necessary attributes and methods to define a relational database. The *Attribute* class stores the information for the definition of each column in relational tables like name, type, and properties. Instances of this class are stored in the *Relation* class to define a relation. Since a relation object is returned from the execution of each DFQL operator in the query, this class contains the necessary attributes to store the parsing and SQL translation information in it. The *database* class stores the relation objects to build the relational database. It also has the necessary information about the back-end RDBMS to which it will be connected since the *database* class is responsible for establishing the connection to the back-end and retrieval operations. Each database connector classes has its child class for each individual back-end RDBMSs. These child classes store information like allowed types, aggregation functions etc. to be used by Amadeus. The class hierarchy of the database connector is depicted in Figure 5.5.

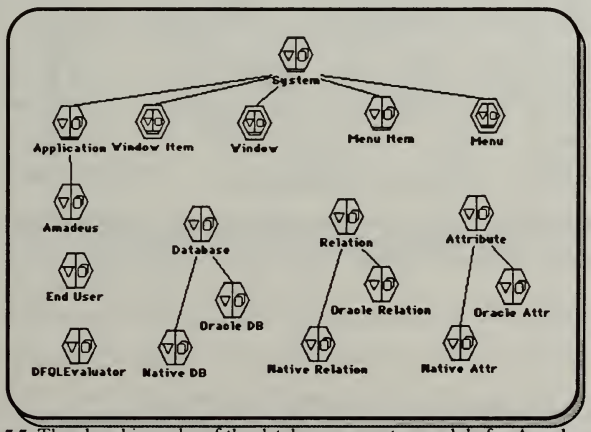


Figure 5.5 The class hierarchy of the database connector module for Amadeus.

A good example of the *extensibility* feature of OOPL is seen here, when it is desired connect more back-ends into the prototype. By inheriting the necessary classes for that

RDBMS, and implementing the specific methods in the child classes directly related to that back-end. All the specific implementations are encapsulated in those individual classes, ensuring the newly connected back-end will function. The OO paradigm is reliable in integrating the classes and using them, the extensibility of the prototype is assured.

The related child classes of the class *relation* have all the methods that are implemented for DFQL primitive operators. Because each primitive operator has a direct execution through the database connector that can be translated to SQL language of that specific RDBMS directly, these methods can access the back-end, execute the operator and get the result. The methods of the class *Oracle Relation* are provided as an example, illustrated in Figure 5.6. Some error checking methods used during the parsing operation will be explained in the next section.

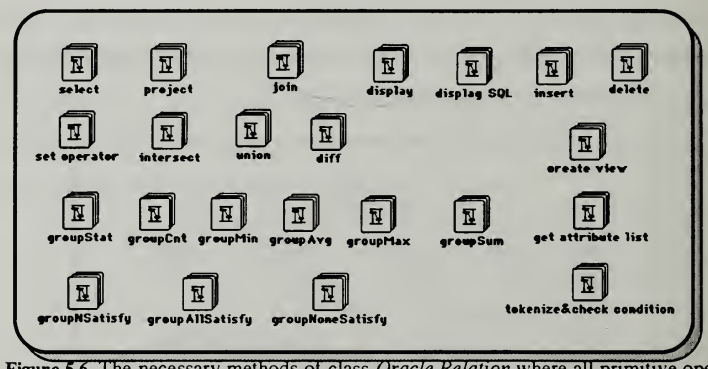


Figure 5.6 The necessary methods of class *Oracle Relation* where all primitive operators are implemented for the Oracle RDBMS.

These methods perform the partial execution and the SQL translation of the optimized query. They are called by DFQL objects during traversing of the query and must return a relation back to the query. Two kinds of query execution exist in this prototype, namely, partial and complete execution. There are two main cases of these methods, named with the names of primitive operators. One of them is for partial execution which takes the necessary inputs, creates the resultant relation object and passes the SQL command to the

back-end to create a similar temporary view in the RDBMS. Hence, the query can be executed up to a specific point and the result e displayed, since the necessary temporary views are created during the execution of previous operators. For the complete execution, however, rather than sending the SQL commands to the back-end for each operator to create temporary views, the optimized SQL translation of the query is sent to the back-end.

Some complex primitive operators are defined in terms of other simple primitive operators simply by calling them in correct order. The implementation of the DFQL primitive operator *groupAllSatisfy* is implemented in this way and is illustrated in Figure 5.7. This notation is explained in Chapter III (Universal Quantification on page 41), using the counting function of the query language.

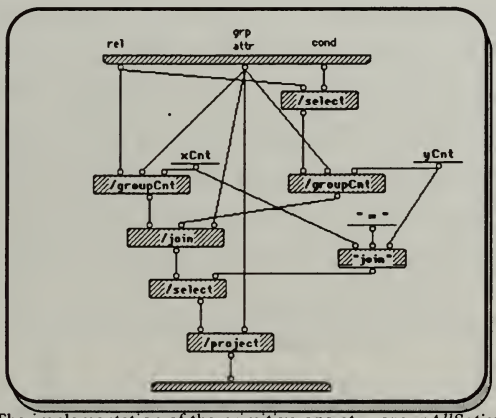


Figure 5.7 The implementation of the primitive operator *groupAllSatisfy* in terms of other simple primitive operators in the class *Oracle Relation*.

D. SQL TRANSLATION

The translation of a DFQL query into SQL is a very important part of this implementation. The features of SQL and DFQL, have been discussed previously (see Chapter II and Chapter III respectively). Since SQL has a declarative nature and DFQL has a procedural nature it is very difficult to translate a DFQL query (a procedural language) to

SQL (a declarative one). Amadeus is designed to use the native language (individual dialects of SQL) of the back-ends instead of using a kernel language. This results in a performance gain for the back-ends during execution. Two types of SQL translation, *partial translation* and *complete translation* are available depending on to the execution methods of query editor. However, a discussion of traversing the data flow query according to queries' formulations to execute the operators is presented first.

1. Traversing the Data Flow Query

Traversing the query is necessary for two purposes. The first reason is to build an optimized complete SQL query called "*parsing*". The other reason is to execute the DFQL objects, one by one, for partial execution, according to their dependencies. Traversing can start from a *display* operator or from an operators' root by double clicking to start the partial execution to that point. Every DFQL object has a method called *runObj* (except similar objects) used during the traversal, as illustrated in Figure 5.8. These methods check the terminals of the object for availability of data. If all of the terminals have their inputs ready, then that object can be executed. Since *DFQLParameter, DFQLRelation*, and *DFQLForm* have no terminals, they can be fired any time. Their methods simply return attribute list, condition string, form, or relation objects to the other operators.

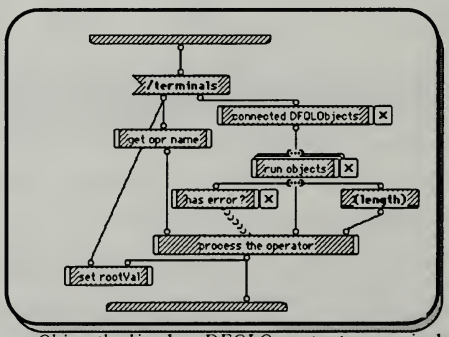


Figure 5.8 The *runObj method* in class *DFQLOperator* to recursively call the same method for traversing the data flow diagram and process the DFQL objects according to their connections.

The illustrated method (in Figure 5.8) belongs to the class called *DFQLPrimOpr* which plays an important role here. This method gets the terminals of the operator, finds the connected DFQL objects, and calls the same method for each connected object recursively. If one of the terminals is an operator, it continues to call other objects recursively until it gets back data. After execution of the methods for each object, error checking is done to look for errors. After this, original operator implemented in the database connector of the connected back-end is called by passing all necessary information along. Since every operator has to return a *relation* object, these results are stored in the operator objects' attribute called *rootVal.* This prevents traversing the same part of the query over and over if an operator's result is fed to more than one terminal.

Polymorphism is used here to recursively call the same method for different objects. While getting the connected objects of the operator, a syntax check of the operators connections is performed, since some terminals of some operators may not be connected properly. For example, if no condition is given to a *join* operator, it must be interpreted as a *cartesian product.* A semantic check of the operator connections checks that each operator's terminals is connected to the correct DFQL object. For instance, the second terminal of a *select* operator can be connected only to DFQLParameter, whereas the third terminal of the *join* operator can be disconnected. These error checking procedures are used for every DFQL object to enforce the formulation of error-free queries.

The traversal is very simple for the user-defined operators, because they have other primitive or user-defined operators as their constituents. The *runObj* method for this type of operator finds the connected DFQL object connected to the user-defined operator's root, and calls the same method for that object. This can easily be done even through other user-defined operators are used in the formulation of the current operator.

2. Partial Translation

This translation is used only for partial execution of the query for debugging purposes. Temporary views are used for each executed operator to store the resultant relation in the back-end. These temporary views can be used in subsequent operators. This is the easiest way of executing the query partially, rather than resorting to complete translation. The created temporary views are deleted from the back-end after each partial execution. As seen in Figure 5.9, if the user wants to execute the query up to the end of the *join* operator, then *temp1* is created for the definition of the *select* operator. *Temp2* is then created by using that temporary view. and then *temp2* can be retrieved and displayed to the user. Since it is difficult to determine the user's behavior during debugging which sections will be executed, this implementation seems sufficient for this purpose.

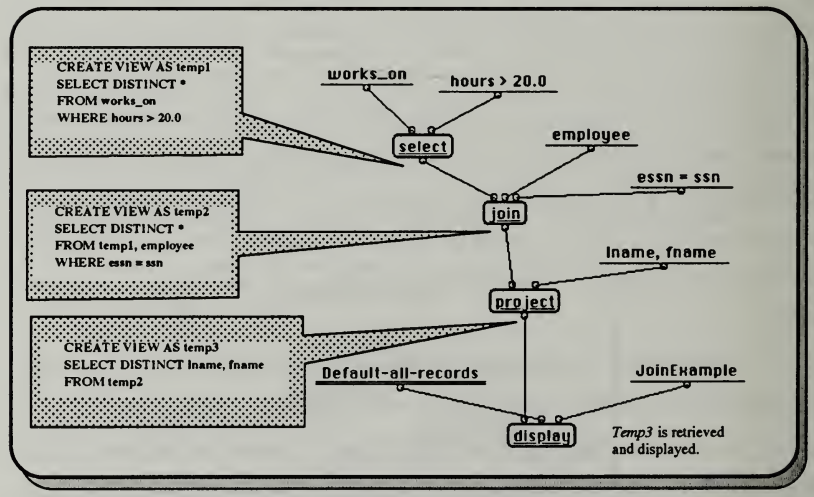


Figure 5.9 The SQL translation during partial execution of DQFL query that is given as (Give the name of employees who work more than 20 hours on a project. on page 23).

3. Complete Translation

This translation is performed when the query is converted to a compact type. Since modification of the query is not allowed, the complete translation of the query can be defined and used to run the query. This form of query permanently saves the definition, making it available for use in applications later on.

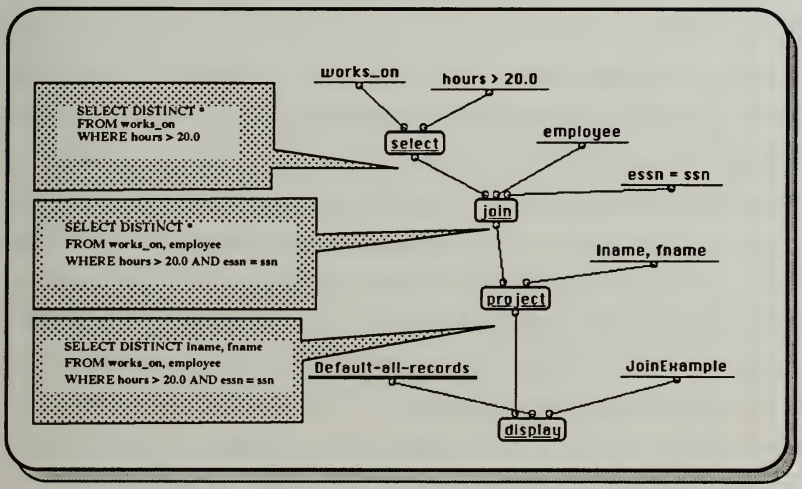


Figure 5.10 The SQL translation for the complete execution of DQFL query given as (Give the name of employees who work more than 20 hours on a project. on page 23).

The translation is done incrementally until some specific conditions are encountered as shown in Figure 5.10. The steps of translation are fairly clear in that figure. Some exceptions of the integration of previous SQL translation and the operator being translated are that if an SQL translation is presented instead of a table, it is not possible to embed the definitions of some operators in that translation. For example, if two SQL translations for a *join* operator are presented instead of two table names, then one cannot be embedded in the other definition(2). A temporary view as discussed in the previous

(2) Nested SQL statements are not desired under these conditions, since they can be used in only a few occasions and they decrease the performance during the back-end process.

section is created for one of them and the view name is embedded into the other definition. Translation after a grouping operator is not allowed, because the *group by* clause creates an entirely different table when used in SQL. Therefore, the same solution is used under these conditions during translation. Using temporary views is not the best solution to translate a DFQL query to SQL, but this seems the only translation technique working correctly for now.

E. USER INFORMATION

A separate class called *EndUser* was created in this design that takes care of user information such as the user-defined operators, user's login names and passwords for the back-ends. This type of information is loaded from a file each the time user runs the prototype. As mentioned before, the *Database Administration Module*, responsible for user information and access rights is not implemented. An instance of the *EndUser* class is stored as an attribute in the class called *Amadeus* which performs all the application operations. The instance of *EndUser* is stored to a disk automatically before quitting the application with a file extension name *UsrInfo*.

VI. CONCLUSIONS

A. SUMMARY

In recent years a broad variety of commercial RDBMSs have become available to the user. All of them use a dialect of SQL for a query language, and are incompatible with each other. If there are several of these products being used in a company in different departments, it is difficult to join them in a federated database system or to share or transfer data between the individual RDBMSs. Using different SQL query language for each database is also very hard for the employees, since some conventions allowed in one RDBMS are not necessarily allowed in another RDBMS.

The purpose of this thesis is to implement a front end system called *Amadeus*, and use the RDBMSs as back-ends (see Figure 6.1) communicating with their own dialect of SQL through the front end. A new query language was developed that eliminates the disadvantages of SQL, and by using the front end system, the same RDBMSs can still be used. *Data Flow Query Language (DFQL)* is implemented in Amadeus, which is based on the dataflow paradigm and has many advantages over SQL.

B. CONCLUSIONS

Amadeus is implemented with DFQL to provide the advantages of the system discussed in Chapter IV. The user can define a database and its relations, manipulate them using the DFQL query language, and retrieve data from the connected RDBMS. DFQL has been proven to be a workable query language with many benefits over the current SQL. It provides many advantages (see Chapter Three) to the user to enforce error-free definitions of queries. Its procedural nature allows the user to express details very easily, including universal and existential quantification. It allows the user to abstract the details into user-defined operators, using them as desired. The debugging features of DFQL, namely

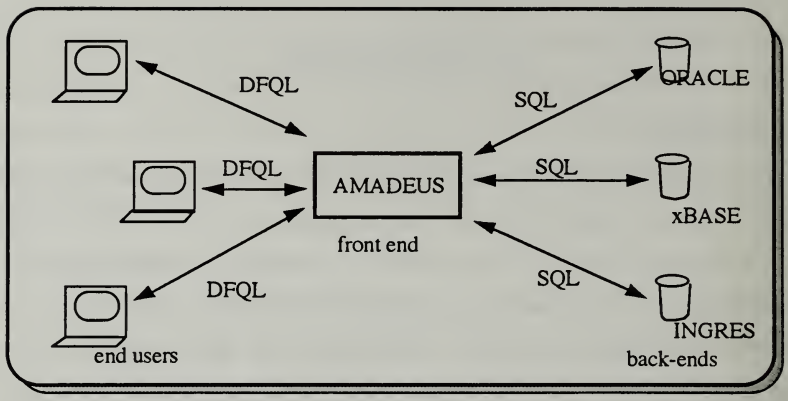


Figure 6.1 The working diagram of *Amadeus* that can communicate with RDBMSs as back-ends.

incremental execution and construction are very useful when the formulation contains semantic errors.

Amadeus eliminates the problems of using different RDBMSs simultaneously and can transfer stored data from different databases. Its object oriented design provides many advantages, like extensibility, modifiability, and maintainability. The number of back-end RDBMS can be increased easily by including the *database connector* containing the necessary classes for the connection. It is easily alterable using the *encapsulation* and *abstraction* features of the object-oriented programming. Amadeus also gives the user many advantages in its interface module, allowing him to define customized input/output forms in which the user can see the results.

C. FUTURE RESEARCH

There is still work to do in Amadeus in its various modules (see Chapter IV). Since all modules designed to be included in this system are not implemented, the capability of Amadeus is currently limited.

Future research areas of the Amadeus prototype system are:

- Integrate the *Interface Module* in the current implementation of Amadeus
- Including more *Database Connectors* to reach more RDBMSs as back-ends
- To extend the translation of DFQL to non-relational database query languages in order to reach to those DBMSs as well
- To implement a *Program Editor* like *Prograph* to define programs and applications that can use current modules
- To implement a *Database Administration Module* to maintain the secrecy and integrity of the data for a multi-user environment and allow the propagation of the definitions according to their access rights
- Design and implement a *Conceptual Design Module* to define the applications in a model and translate them automatically to the relational model
- Establish a network connection and using secure network protocols to reach the back-ends located in another computer

These primary research areas will increase the capability of the Amadeus system. Some are currently difficult to design and implement, but improvement in software development will provide more convenient languages and tools to complete their design and implementation easily in the future.

LIST OF REFERENCES

[Andyne91] Andyne Computing Limited, GQL: *Graphical Query Language; GQL/User Demo Guide,* Kingston, Ontario, March 1991.

[Angelaccio90] Angelaccio, M., Ctarci, T., and Santucci, G. *QBD: Graphical Query Language with Recursion,* IEEE Transactions on Software Engineering, v. 16, pp. 1150-1163, October 1990.

[Chen76] Chen, P. *The Entity Relationship Mode-- Toward a Unified View of Data,* TODS, March 1976.

[Clark91] Clark, Gard, and Wu, C.T., *Dataflow Query Language for Relational Databases,* Department of Computer Science Naval Postgraduate School, Monterey CA.

[Czejdo90] Czejdo, B., *A Graphical Data Manipulation Language for an Extended ER Model,* IEEE Computer, v.23, pp. 26-36, March 1990.

[Dadashzadeh90] Dadashzadeh, M., and Stemple, D., "*Converting SQL queries into relational algebra*" Information & Management, v. 19, pp. 307-323, December 1990.

[Elmasri89] Elmasri, R. and Navathe, S., *Fundementals of Database Systems,* Benjamin/Cumming Publishing Company, 1989.

[Hargrove93] Hargrove, James Phillip and Wu, C.T. , *Design And Implementation of an Interface Editor for the Amadeus Multi-Relational Database Front-end System,* Department of Computer Science Naval Postgraduate School, Monterey CA.

[TGS88a] The Gunakara Sun Systems, *Prograph Tutorial,* 1988

[TGS88b] The Gunakara Sun Systems, *Prograph Reference, 1988.*

[TGS90] The Gunakara Sun Systems, *Prograph 2.0 Technical Specifications,* 1990.

[TGS91] The Gunakara Sun Systems, *Prograph 2.5 Updates, 1991.*

[Wong82] Wong, H. K. T., and Kuo, I., *GUIDE: Graphical User Interface for Database Exploration* Proceedings of the Eighth International Conference on very Large Databases, pp. 22-32, September 1982.

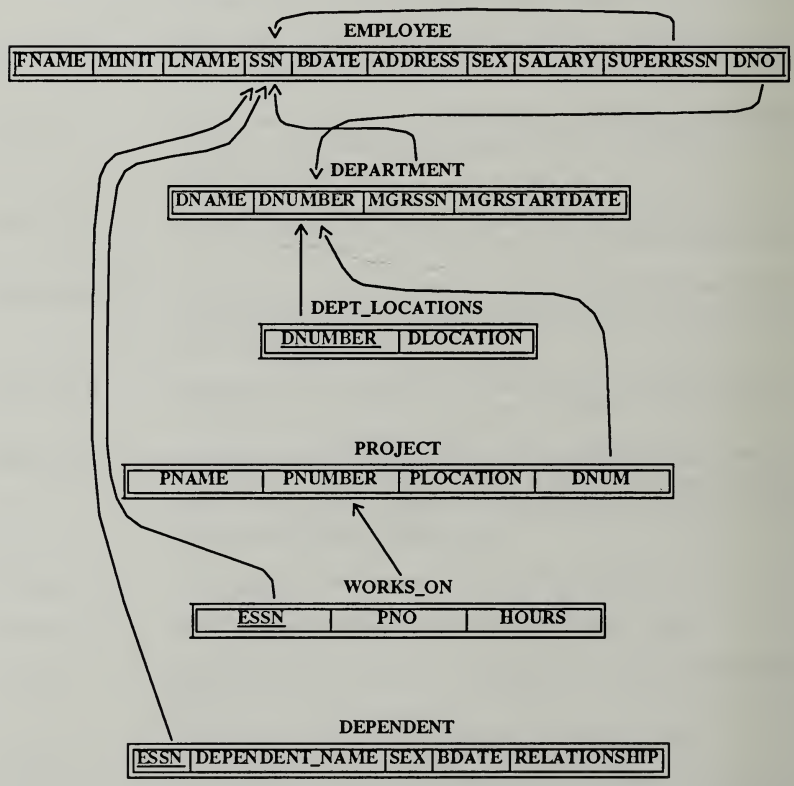
[Wu91a] Wu, C.Thomas, and Clark Gard J., *DFQL: Dataflow Query Language for Relational Databases*, Department of Computer Science Naval Postgraduate School, Monterey CA.

[Wu91b] Wu, C.T., *OOP + Visual Dataflow Diagram = Prograph,* Journal of Object Oriented Programming, pp 71-75, June 1991.

APPENDIX A

SAMPLE DATABASE

All queries and examples are built from this relational database example called *Company Database*. [Elmasri89] throughout this thesis.



The arrows shows the references established by foreign keys. In order to use this database to build some example queries, some values are entered, as shown in tables.

EMPLOYEE

NAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERRSSN	DNO
John	B	Smith	123456789	09-JAN-55	731 Fondren, Houston, TX	M	30000	333445555	5
ranklin	T	Wong	333445555	08-DEC-45	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	19-JUL-58	3321 Castle, Spring, TX	F	25000	987654321	4
ennifer	S	Wallace	987654321	20-JUN-31	291 Berry, Bellaire, TX	F	43000	888665555	4
amesh	K	Narayan	666884444	15-SEP-52	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	31-JUL-62	5631 Rice, Houston, TX	F	25000	333445555	5
hmad	V	Jabbar	987987987	29-MAR-59	980 Dallas, Houston, TX	M	25000	987654321	4
ames	E	Borg	888665555	10-NOV-27	450 Stone, Houston, TX	M	55000	null	1

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research	5	333445555	22-MAY-78
Administration	4	987654321	01-JAN-85
Headquarters	1	888665555	19-JUN-71

DEPT_LOCATIONS

DNUMBER	DLOCATION
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM
ProductX	2	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

WORKS_ON

ESSN	PNO	HOURS
123456789	1	32.5
123456789	2	7.5
666884444	1	20.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	1	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987987987	30	20.0
987987987	20	15.0
888665555	20	null

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
333445555	Alice	F	05-APR-76	DAUGTER
333445555	Theodore	M	25-OCT-73	SON
333445555	Joy	F	03-MAY-48	SPOUSE
987654321	Abner	M	29-FEB-32	SPOUSE
123456789	Michael	M	01-JAN-78	SON
123456789	Alice	F	31-DEC-78	DAUGHTER
123456789	Elizabeth	F	05-MAY-57	SPOUSE

The rest of this section is filled out with some result values of the queries that are given as examples in Chapter III. These results are taken from Query editor by executing the same queries.

Table 5: QUERY 3.2

dname
Headquarters

Table 6: QUERY 3.3

ame	minit	lname	ssn	bdate	address	sex	salary	superssn	dno
mad	V	Jabbar	987987987	November 25, 1957	980 Dallas, Houston, TX	M	25000	987654321	4
nklin	T	Wong	333445555	April 23, 1956	638 Voss, Houston, TX	M	40000	888665555	5
nes	E	Borg	888665555	February 11, 1921	450 Stone, Houston, TX	M	55000	NULL	1
n	B	Smith	123456789	October 22, 1955	731 Fonren, Houston, TX	M	30000	333445555	5
mesh	K	Narayan	666884444	March 20, 1963	975 Fire Oak, Humble, TX	M	38000	333445555	5

Table 7: QUERY 3.4

lname	fname	salary	address
Borg	James	55000	450 Stone, Houston, TX
English	Joyce	25000	5631 Rice, Houston, TX
Jabbar	Ahmad	25000	980 Dallas, Houston, TX
Narayan	Ramesh	38000	975 Fire Oak, Humble, TX
Smith	John	30000	731 Fonren, Houston, TX
Wallace	Jennifer	43000	291 Berry, Bellaire, TX
Wong	Franklin	40000	638 Voss, Houston, TX
Zelaya	Alicia	25000	3321 Castle, Spring, TX

Table 8: QUERY 3.5

averageHours
17.1875

Table 9: QUERY 3.6

lname	fname
Jabbar	Ahmad
Narayan	Ramesh
Smith	John
Zelaya	Alicia

APPENDIX B

TERMINOLOGY OF PROGRAPH

A. LANGUAGE BASICS

1. Pictorial Representation of the Language

Prograph programs are composed entirely of icons and amplifying text. Table 10 shows common icons used in constructing Prograph programs.

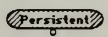
SYMBOL	EXPLANATION	SYMBOL	EXPLANATION
	This symbol stores the constant values like integers, strings, and lists.		This is a simple operator that contains the methods of classes.
	This is used to read a value of an attribute in any class.		This is used to store a value into an attribute in any class.
	This is used to create a new instance (object) of a class to be used in program.		This allows to get/set a value from/to persistent storage of the language.
	This encapsulates some other methods like a subprogram.		This performs a calculation of its inputs **a** and **b** and gives the result back.
	This is a loop that carries the result to subsequent iterations.		This performs the same method to each item of given list.

Table 10: EXAMPLES OF *PROGRAPH* PROGRAMMING LANGUAGE SYMBOLS

2. Control Structures

Prograph *Control Structures* control the flow of execution within a program. Control structures are composed of icons (either an 'X' or a '√') that are attached to the right-hand side an operator, and are activated on either the success or failure of the associated operation.

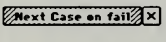
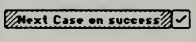
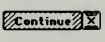
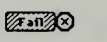
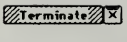
SYMBOL	EXPLANATION	SYMBOL	EXPLANATION
	If the incoming data does not match then fire next case.		If the incoming data matches then fire next case.
	If method fails during execution then fire the next case.		If method runs without failing then fire the next case.
	If method fails during execution then continue on this case.		If method fails then make this method to fail too.
	If method fails then terminate the execution.		If method fails then finish the iteration and stop execution.

Table 11: EXAMPLES OF *PROGRAPH* PROGRAMMING LANGUAGE CONTROL SYMBOLS

The default control structure is *success*. Operations *fail* in one of three ways in a match operation:

(1) The items being compared do not match,

(2) A Boolean operation returns a FALSE value, or

(3) A *FAIL* condition is propagated to a particular operation.

Operations may also generate errors under certain conditions, including: type mis-matches, syntax errors, or a specific program condition which cannot be satisfied by

the particular control structure. Table 11 shows typical Prograph control structures. An 'X' within a control structure indicates that it is activated if the associated operation fails. A check mark (√) indicates that the control structure is activated if the associated operation succeeds. Other graphics inside the control structure icon indicate additional action to be taken.

The most basic Prograph conditional execution format is the *Next Case* with an accompanying match operation or conditional test. Figure B. 1 depicts a conditional test with a *match on success* control structure which tests for a specific condition to determine which of two case windows will be executed.

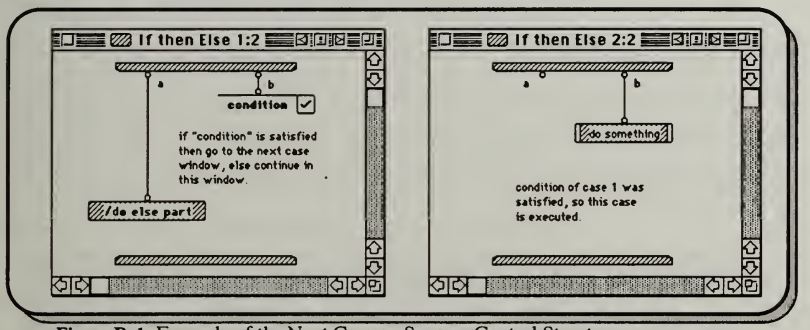


Figure B. 1 Example of the Next Case on Success Control Structure

3. Classes and Inheritance

Classes of objects, and all inheritance relationships, appear on the screen as trees of icons. The Prograph class system provides a means for constructing a new class from an existing class through inheritance. A Prograph class can inherit from at most one parent. Multiple inheritance is not currently allowed in this language.

The class icon is a hexagon which is divided into two parts: *attributes* on the left, and *methods* on the right. Double-clicking on the left half of a class icon displays the attributes of the class, while double-clicking on the right half displays the class methods. The class hierarchy and inheritance links are shown in Figure B. 2.

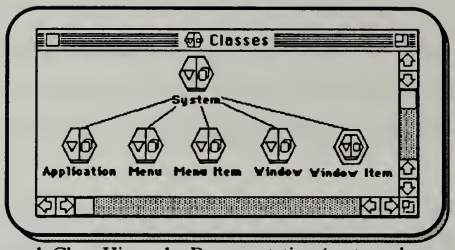


Figure B. 2 Prograph Class Hierarchy Representation (system classes are shown.)

4. Attributes

Prograph attributes are displayed in an *Attributes Window*. There are two types of Prograph attributes: *instance* and *class*. An instance attribute may have a different value for each instance of a class. Class attributes, however, have one value for the class as a whole. Therefore, the value of a class attribute is shared by all instances of the class. The attribute icon is a downward pointing triangle. A typical attribute window of a class is shown in Figure B. 3.

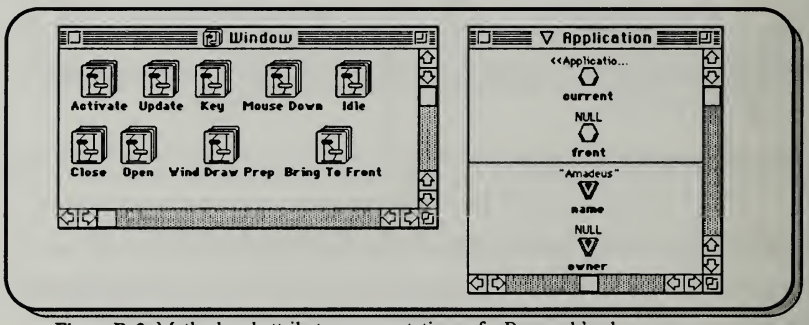


Figure B. 3 Method and attribute representations of a Prograph's class.

5. Methods and Cases

A Prograph method consists of a sequence of one or more dataflow, called *cases*. A case consists of an input bar, an output bar, operations and datalinks, Data flows into a case via the input bar, and out through the output bar.

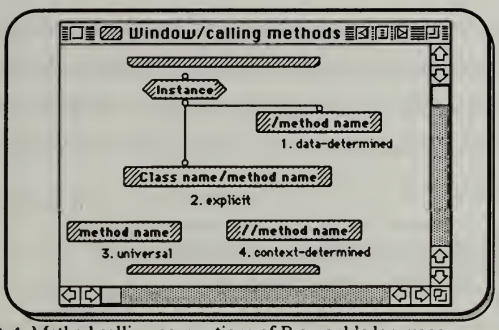


Figure B. 4 Method calling conventions of Prograph's language.

Methods are referenced in one of four ways: *universal, data-determined, explicit and context-determined* (see Figure B. 4). These terms correspond to the terms global, regular, early-bound and self, which are more commonly used in object-oriented programming literature ([Wu91b] p. 71). Essentially, the calling format determines where Prograph looks for the referenced method in the class hierarchy.

- Data-determined. Prograph looks for the referenced method in the class of the object which flows into the leftmost terminal of the method.
- Explicit. Prograph looks for the referenced method in the class which is explicitly listed to the left of the "/" in the method icon. If the method is not found in the explicitly listed class, then Prograph uses inheritance links to check ancestor classes for the method.
- Universal. This is a call to a global method.
- Context-determined. Prograph looks for the referenced method in the same class as the current method that contains the method referencing operation. This allows a method to send a message to itself.

6. Operations

An operation is the basic executable component of a case. Operations have a name, zero or more inputs, zero or more outputs and a distinctive icon. Data flows into an operation through terminals located on the top of the operation icon, and out through roots located on the bottom of the icon. Prograph provides a special icon, called a *synchro link* which forces a specific execution *order* on a pair of operations (see Figure B. 5).

However, the synchro link does not guarantee that the operations will execute consecutively, only that one will execute before the other. ([TGS88b] p. 7) In the example shown below, number one will execute before number two. However, there is no guarantee that number two will execute *immediately* after number one, since there is no way to determine when number three will execute.

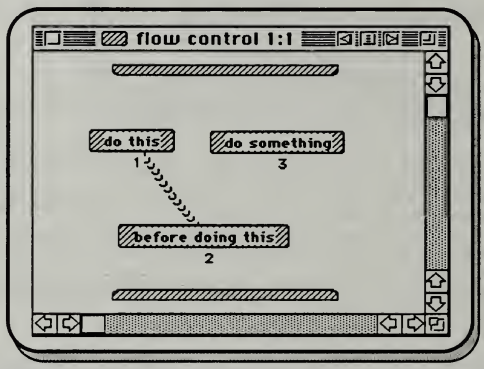


Figure B. 5 Synchro Link to control the execution order of the methods in Prograph.

7. Message Passing

Message passing in Prograph is similar to most other object-oriented languages. Some differences occur, however, because of the dataflow nature of the Prograph language. Essentially, in Prograph objects flow into operations to initiate actions. In a "standard" object-oriented programming language, a stationary object sends a message to another stationary object. Although the models are somewhat different, the basic concepts are the same. ([TGS88a] p. 93)

8. Primitives

Prograph primitives are calls to compiled methods, and are categorized into sixteen groups, including: Application, Bit, Data, File, Graphics, Instances, Interpreter Control, I/O, Lists, Logical/Relational, Math, Memory, Strings, System, Text and Type.

Primitives comprise the kernel of Prograph's functionality. Unlike other object-oriented programming languages, Prograph primitives do not belong to any class. This, and the fact that the language supports regular data types such as *string*, *integer*, *Boolean* and *real* make Prograph a hybrid object-oriented programming language. ([Wu91b] p. 72)

B. THE PROGRAPH ENVIRONMENT

The Prograph language is seamlessly integrated with the Prograph development environment. An editor provides a visual interface for creating and modifying programs, while an interpreter contains features which allow dataflow diagrams to be displayed during execution, in effect graphically animating the flow of data throughout a program as each operation is executed ([TGS90] p. 21).

1. Editor

The Prograph editor is context sensitive, so syntax errors are caught at the time they are created, eliminating the need for a traditional debugger. During program execution, run-time errors are flagged, program execution is halted and the appropriate dataflow diagram displayed. This enables the user to correct the error and immediately resume execution. An on-line help system is also available and is fully integrated into the editor.

2. Interpreter

The Prograph interpreter is highly interactive. Program execution may be paused at any point and dataflow diagrams and data values examined, allowing simultaneous execution and editing of applications. Additionally, program execution may be traced step by step, allowing the flow of data through a program to be traced visually. If a dataflow diagram is changed while execution is paused, the interpreter backs up to the change and continues execution from that point.

C. COMPILER

The Prograph compiler generates stand-alone application programs, and allows linking to modules developed with other programming languages such as MPW C™ and Think C™. The compiler also includes an intelligent *Project Manager* which keeps track of the files needed to build a particular application. The Project Manager selects only the code actually required when building a stand-alone application and informs the user of any missing code. If the compiler detects an error in a Prograph file, the user can enter the editor/interpreter to see the operation that generated the error.

A certain amount of overhead is normally introduced when creating stand-alone applications. In Prograph, stand-alone applications which do not use system classes require an additional 50Kbytes of overhead, while those with system classes require an additional 130Kbytes. However, the execution speed of compiled Prograph code is, on the average, 15 times faster than the same interpreted code ([TGS90] p. 33-36).

APPENDIX C

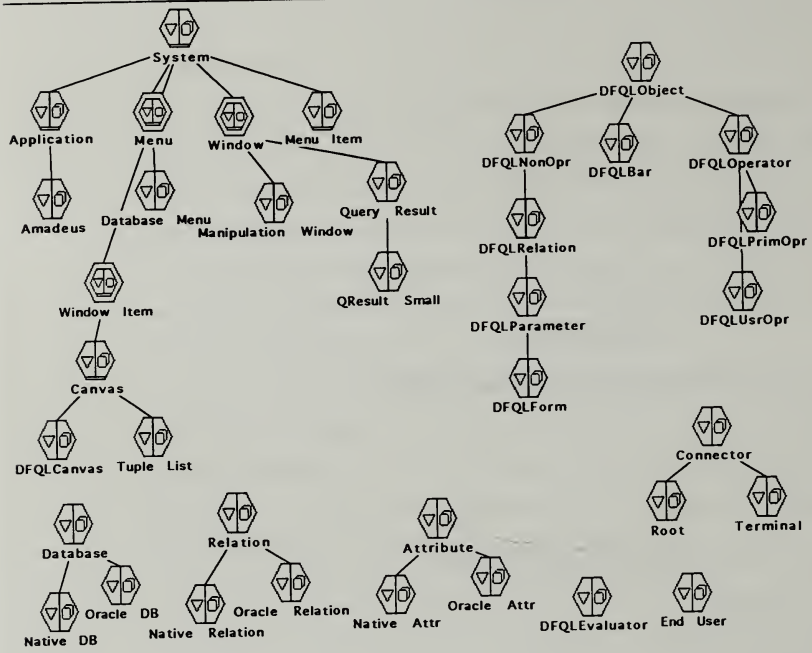
SOURCE CODE FOR AMADEUS

(1) Class hierarchy.

(2) Important Class Methods and Attributes[1]:

- Amadeus
- Oracle Relation
- Oracle DB
- DFQL Relation
- DFQLUsrOpr
- DFQLPrimOpr
- DFQLObject
- DFQLCanvas
- Manipulation Window

(1)Local methods are not included for simplicity.


System
Application
Menu
Window
Menu Item
Amadeus
Database
Menu Manipulation
Window
Query Result
Window Item
QResult
Small
Canvas
DFQLCanvas
Tuple
List
DFQLObject
DFQLNonOpr
DFQLBar
DFQLOperator
DFQLRelation
DFQLPrimOpr
DFQLParameter
DFQLUsrOpr
DFQLForm
Connector
Root
Terminal
Database
Native DB
Oracle DB
Relation
Native Relation
Oracle Relation
Attribute
Native Attr
Oracle Attr
DFQLEvaluator
End User

∇ Amadeus

<<Amadeus>>

current

N.I.L.

front

""

same

N.I.L.

owner

FALSE

active?

N.I.L.

menu bar

N.I.L.

resources

"/About"

about method

()

servant methods

()

menus

()

windows

()

menu lib

()

window lib

<<End User> <End User>
using this app

current user

Amadeus

init menus, read in sysinfo,
and display opening
dialog

adds a newly created db
to the list of dbs

db

save the sysinfo back
to dbs file

dbs

return all available
primitive operators

oprs

return all available
user defined operators

oprs

returns objs of
current <Relations>

relations

returns a list of
available bckend
DB type names
(e.g. Ingres, Oracle, etc.)

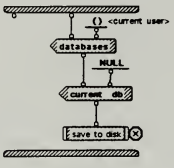
DBs

forms

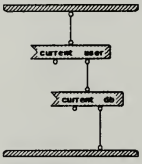
t db

info

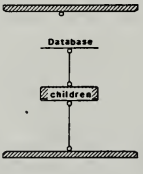
Amadeus/save usr info 1:1



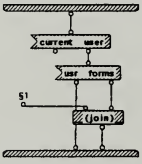
Amadeus/get current db 1:1



Amadeus/available DBs 1:1

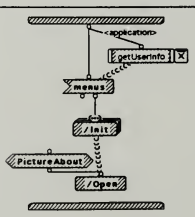


Amadeus/get usr forms 1:1



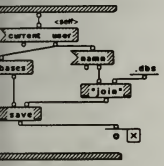
§1. ("Default-one-record" "Default-all-records")

Amadeus/init 1:1

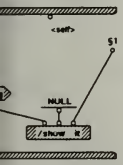


Amadeus/add new db 1:1

Amadeus/save dbs 1:2

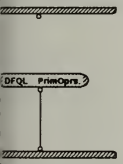


Amadeus/save dbs 2:2



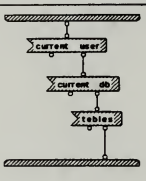
the database information !!

Amadeus/get prim oprs 1:1



Amadeus/get usr oprs 1:1

Rmadeus/get base relations 1:1



Oracle Relation

TRUE = TRUE parse only
= FALSE parse and execute
parse only?

NULL
ecmd

{} list of attributes
attributes

<<Oracle DB: Database object that self belongs to
owner

NULL
sqlStmt

FALSE TRUE if it is a base relation
base relation?

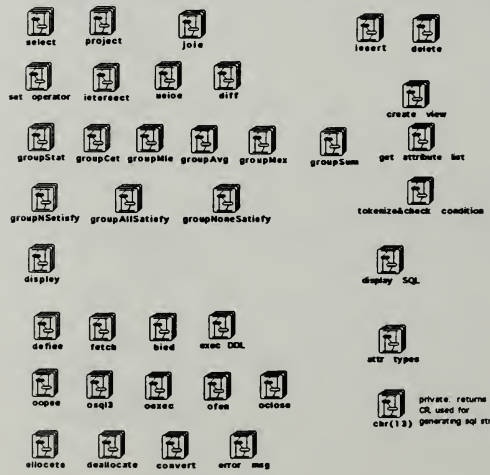
<<DFQLEvaluator stores the DFQLEvaluater for the final relation, otherwise NULL
SQLEvaluation

{} list of retrieved tuples
tuples

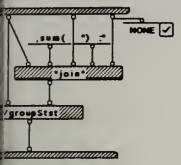
NULL ptr to (actual data area (cda) where information is stored. Allocation only when the sqlStmt is executed.
data area

FALSE =TRUE error has occur in processing oci calls, no further processing will occur
has error?

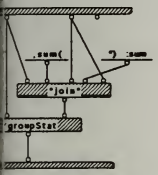
Oracle Relation



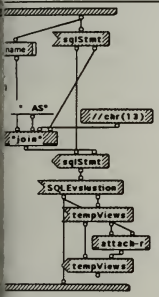
Oracle Relation/groupSum 1:2



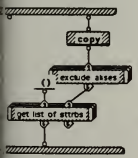
Oracle Relation/groupSum 2:2



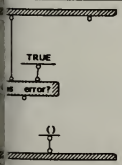
Oracle Relation/create view 1:1



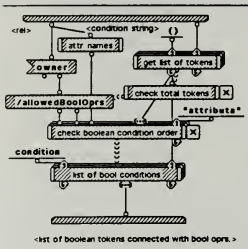
Oracle Relation/get attribute list 1:2



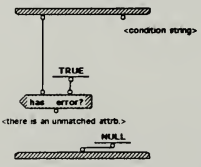
Oracle Relation/get attribute list 2:2



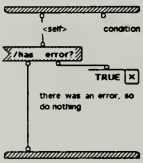
Oracle Relation/tokenize&check condition 1:2



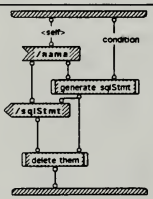
Oracle Relation/tokenize&check condition 2:2



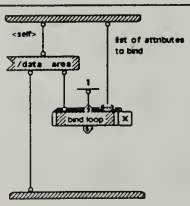
Oracle Relation/delete 1:2



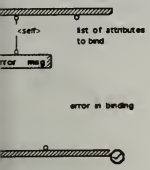
Oracle Relation/delete 2:2



Oracle Relation/bind 1:2



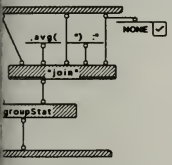
Oracle Relation/bind 2:2



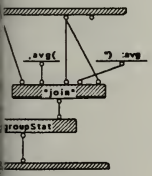
Oracle Relation/diff 1:1

Oracle Relation/union 1:1

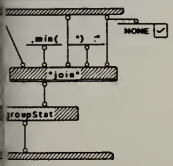
Oracle Relation/groupAvg 1:2



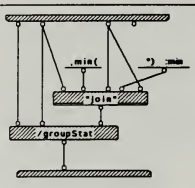
Oracle Relation/groupAvg 2:2



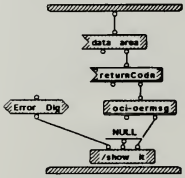
Oracle Relation/groupMin 1:2



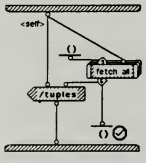
Oracle Relation/groupMin 2:2



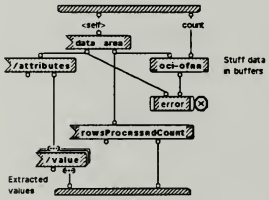
Oracle Relation/error msg 1:1



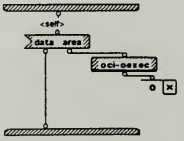
Oracle Relation/fetch 1:1



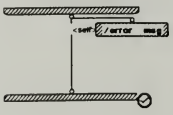
Oracle Relation/ofen 1:1



Oracle Relation/oexec 1:2

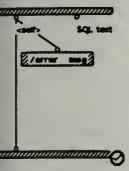


Oracle Relation/oexec 2:2



Oracle Relation/osql3 1:2

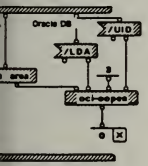
Oracle Relation/osql3 2:2



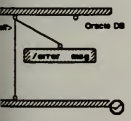
Oracle Relation/oclose 1:2

Oracle Relation/oclose 2:2

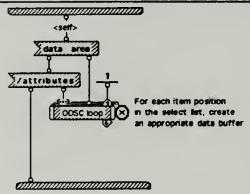
Oracle Relation/oopen 1:2



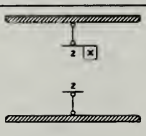
Oracle Relation/oopen 2:2



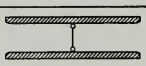
Oracle Relation/define 1:1



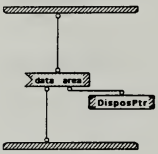
Oracle Relation/convert 1:2



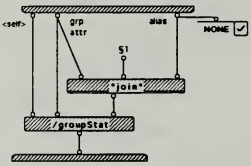
Oracle Relation/convert 2:2



Oracle Relation/deallocate 1:1

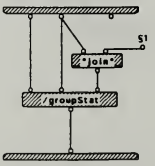


Oracle Relation/groupCnt 1:2



§1. *,count(*) *

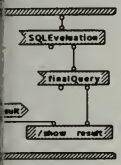
Oracle Relation/groupCnt 2:2



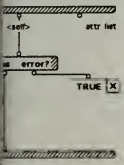
§1 *,count(*) xcnt*

Oracle Relation/intersect 1:1

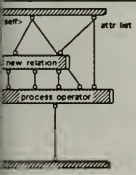
Oracle Relation/display SQL 1:1



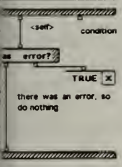
Oracle Relation/project 1:2



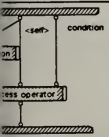
Oracle Relation/project 2:2



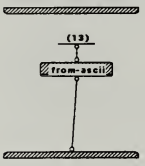
Oracle Relation/select 1:2



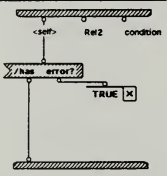
Oracle Relation/select 2:2



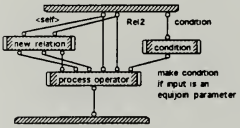
Oracle Relation/chr(13) 1:1



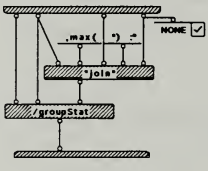
Oracle Relation/join 1:2



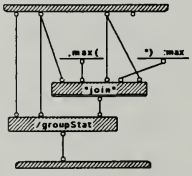
Oracle Relation/join 2:2



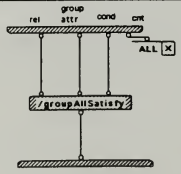
Oracle Relation/groupMax 1:2



Oracle Relation/groupMax 2:2

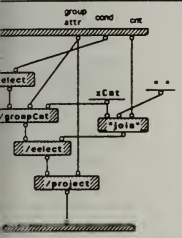


Oracle Relation/groupNSatisfy 1:3

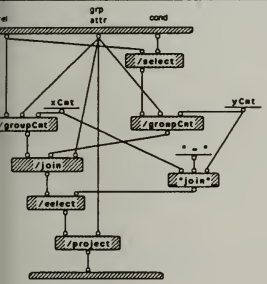


Oracle Relation/groupNSatisfy 2:3

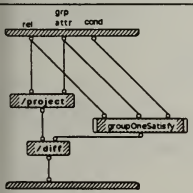
Oracle Relation/groupNSatisfy 3:3



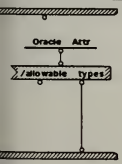
Oracle Relation/groupAllSatisfy 1:1



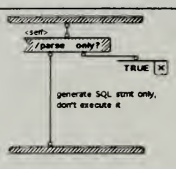
Oracle Relation/groupNoneSatisfy 1:1



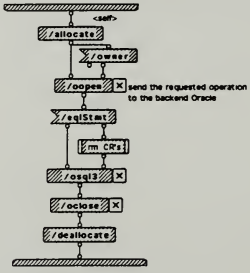
Oracle Relation/attr types 1:1



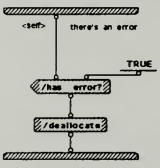
Oracle Relation/exec DDL 1:3



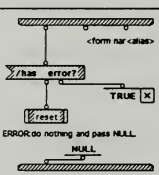
Oracle Relation/exec DDL 2:3



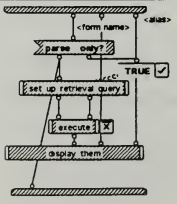
Oracle Relation/exec DDL 3:3



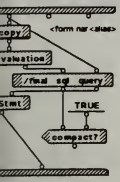
Oracle Relation/display 1:3



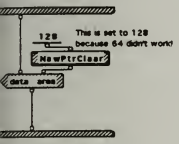
Oracle Relation/display 2:3



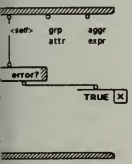
Oracle Relation/display 3:3



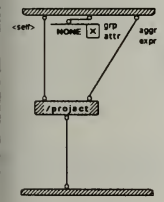
Oracle Relation/allocate 1:1



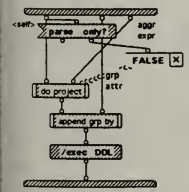
Oracle Relation/groupStat 1:4



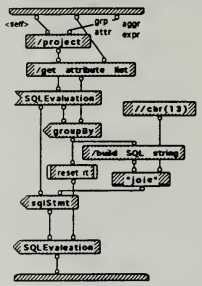
Oracle Relation/groupStat 2:4



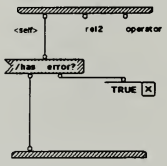
Oracle Relation/groupStat 3:4



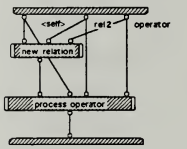
Oracle Relation/groupStat 4:4



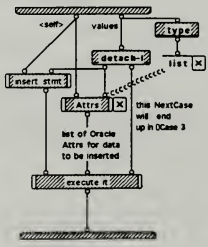
Oracle Relation/set operator 1:2



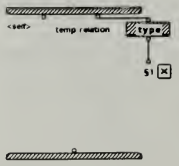
Oracle Relation/set operator 2:2



Oracle Relation/insert 1:3

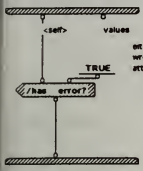


Oracle Relation/insert 2:3



§1 Oracle Relation

Oracle Relation/insert 3:3



▽ Oracle 08

"Oracle" name of backend database

type name

NULL

ecme

NULL

description

( ) list of relations stored in this db

tables

("AVG" "CN" list of supported aggregate functions

aggr functions

NULL ptr to logon data area

lda

NULL ptr to host data area

hda

NULL

eser

NULL

paseword

0 tmp id # for creating temp relations

tempNameId

Oracle 08

LDA

UID

temp ReName

error

create backend

ose ologof obreak ocom orol

add table

nnects to the ck end Oracle

disconnect

reset

reset tmpRelname and drop temp view create in Oracle

calls connect method le compatible with other ises.

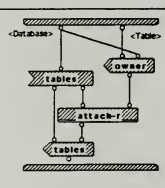
allowedBoolOprs

Oracle 08/create backend 1:1

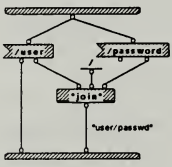
<oracle db>

latter will be implemented to create new databases

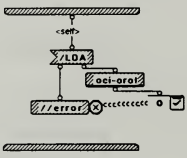
Oracle 08/add table 1:1



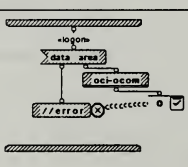
Oracle 08/UID 1:1



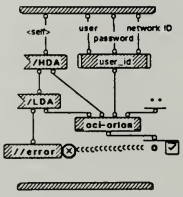
Oracle 08/orol 1:1



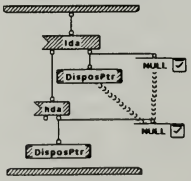
Oracle 08/ocom 1:1



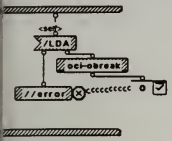
Oracle 08/orion 1:1



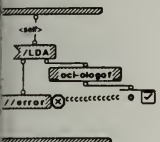
Oracle 08/dispose 1:1



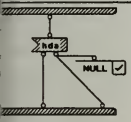
Oracle DB/obreak 1:1



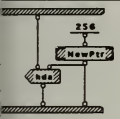
Oracle DB/ologof 1:1



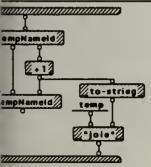
Oracle DB/HDA 1:2



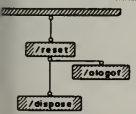
Oracle DB/HDA 2:2



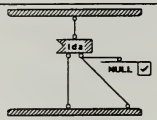
Oracle DB/temp Reiname 1:1



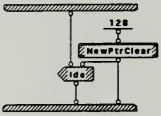
Oracle DB/disconnect 1:1



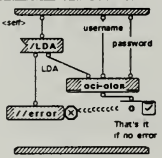
Oracle DB/LDA 1:2



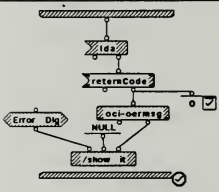
Oracle DB/LDA 2:2



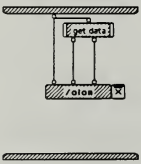
Oracle DB/olon 1:1



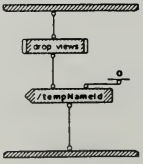
Oracle DB/error 1:1



Oracle DB/connect 1:1

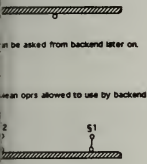


Oracle DB/reset 1:1



Oracle DB/open I:t

Oracle DB/allowedBoolOprs t:t



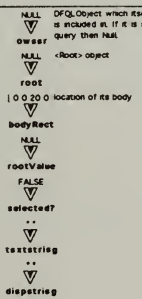
in be asked from backend later on.

ean oprs allowed to use by backend

2 51
)

R" "XOR")
=" ">=" "<=" "/=")

DFQLRelation



DFQLRelation

turns base Relation
presented by this
bject

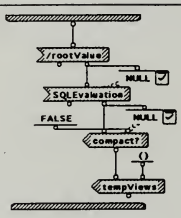
updates the root value with current relation with this name.

obj

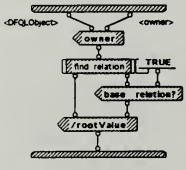
DFQLRelation/reset rootVal t:t

ust a dummy method
do nothing for DFQLRelation.

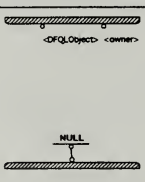
DFQLRelation/uncompact obj 1:1



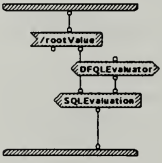
DFQLRelation/put contents 1:2



DFQLRelation/put contents 2:2



DFQLRelation/run obj 1:1



▽ DFQLUsrDpr

owner — NULL — DFQLObject which itself is included in. If it is in query then Null.

root — | 20 0 28 0 — <Root> object

bodyRect — | 0 0 20 0 — location of its body

rootValue — NULL

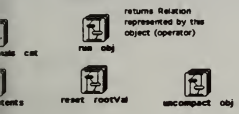
selected? — FALSE

oprname — ••

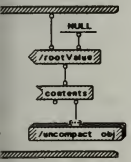
terminals — NULL — list of <Terminal> ~((termract fromObjinstnum) ..)

contents — () — list of the DFQLobjects it is composed of

DFQLUsrOpr



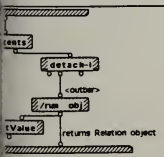
DFQLUsrOpr/uncompact obj 1:1



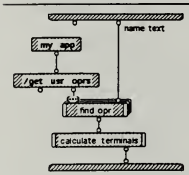
DFQLUsrOpr/put contents 1:1

DFQLUsrOpr/run obj 1:2

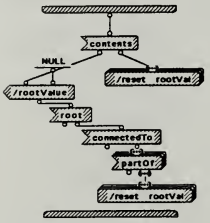
DFQLUsrOpr/run obj 2:2



DFQLUsrOpr/get terminals cnt 1:1



DFQLUsrOpr/reset rootVal 1:1



▽ DFQLPrimOpr

NULL — DFQLObject which itself is included in. If it is in query then Null.
owner

| 20 0 28 0 — <Root> object
root

| 0 0 20 0 — location of its body
bodyRect

NULL
rootValue

FALSE
selected?

""
oprname

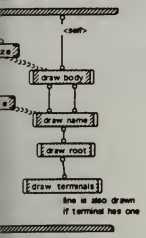
NULL — list of <Terminal> —((termrect fromObjInstnum) ..)
terminals

DFQLPrimOpr

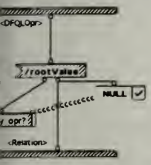
run obj — returns Relation represented by this object (operator)

draw — draw itself on the currently *sc-begin*ed canvas with underline text

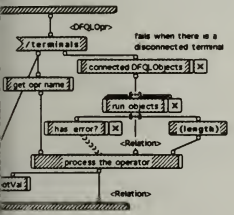
DFQLPrimOpr/draw 1:1



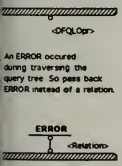
DFQLPrimOpr/run obj 1:3



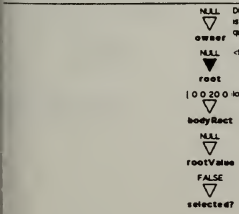
DFQLPrimOpr/run obj 2:3



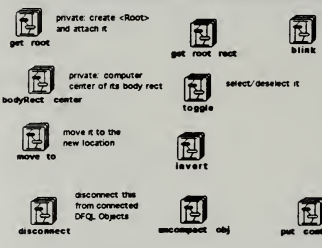
DFQLPrimOpr/run obj 3:3



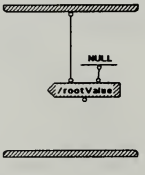
DFQLObject



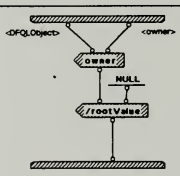
DFQLObject



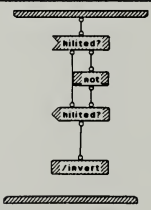
DFQLObject/uncompact obj 1:1



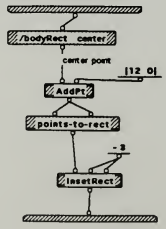
DFQLObject/put contents 1:1



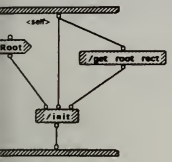
DFQLObject/blink 1:1



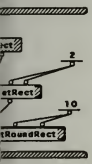
DFQLObject/get root rect 1:1



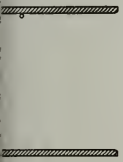
DFQLObject/get root 1:1



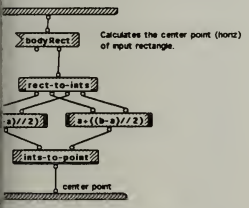
DFQLObject/invert 1:2



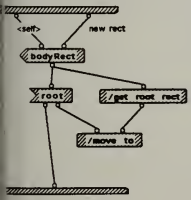
DFQLObject/invert 2:2



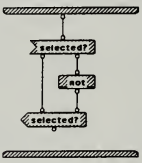
DFQLObject/bodyRect center 1:1



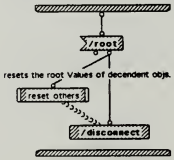
DFQLObject/move to 1:1

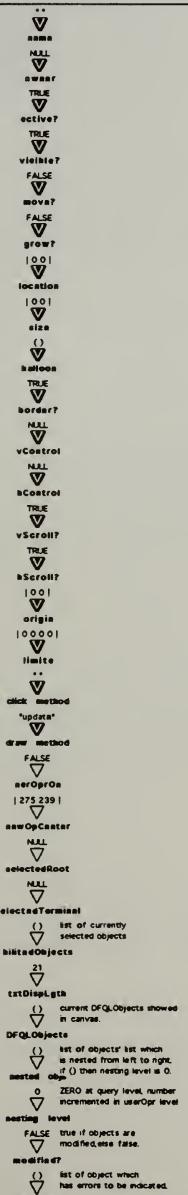


DFQLObject/toggle 1:1



DFQLObject/disconnect 1:1




" "
name
NIL
owner
TRUE
active?
TRUE
visible?
FALSE
move?
FALSE
grow?
| 0 0 |
location
| 0 0 |
size
( )
halfoos
TRUE
border?
NIL
vControl
NIL
hControl
TRUE
vScroll?
TRUE
hScroll?
| 0 0 |
origin
| 0 0 0 0 |
limits
" "
click: method
"update"
draw: method
FALSE
userOprOn
| 275 239 |
newOpCenter
NIL
selectedRoot
NIL
selectedTerminal
( )
list of currently
selected objects
hiliteObjects
21
txtDisplength
( )
current DFQLObjects showed
in canvas.
DFQLObjects
( )
list of objects' list which
is nested from left to right.
if ( ) then nesting level is 0.
nested objs
0
ZERO at query level, number
incremented in userOpr level
nesting level
FALSE
true if objects are
modified,else false.
modified?
( )
list of object which
has errors to be indicated.
blinking objs

DFQLCanvas

process click

create DFQLObject

remove DFQLObjects

remove the selected DFQLObjects, calls erase DFQLObject.

bodyRect

erase line

init draw

erase

clear itself

draw line

end draw

reset

redraw

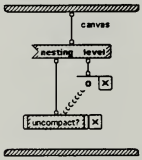
refresh itself

update

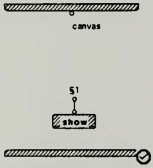
calls redraw, needed for external connection

is allowed?

DFQLCanvas/is allowed? 1:2

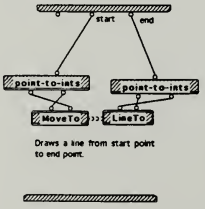


DFQLCanvas/is allowed? 2:2



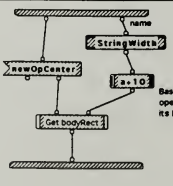
§1. "Modification is NOT allowed !!!"

DFQLCanvas/draw line 1:1



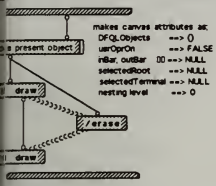
Draws a line from start point to end point.

DFQLCanvas/bodyRect 1:1

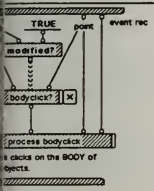


Based on the length of the operator name determines its bodyRect

DFQLCanvas/reset 1:1

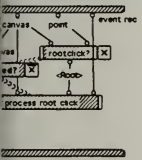


DFQLCanvas/process click 1:4

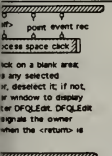


DFQLCanvas/process click 2:4

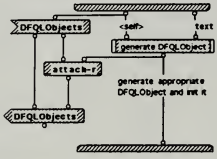
DFQLCanvas/process click 3:4



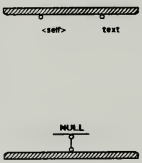
DFQLCanvas/process click 4:4



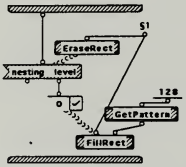
DFQLCanvas/create DFQLObject 1:2



DFQLCanvas/create DFQLObject 2:2

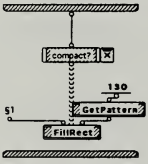


DFQLCanvas/erase 1:2



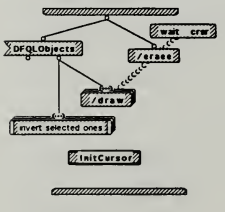
§1. |0 0 500 500|

DFQLCanvas/erase 2:2

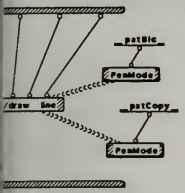


§1. |0 0 500 500|

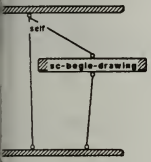
DFQLCanvas/redraw 1:1



DFQLCanvas/erase line 1:1

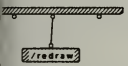


DFQLCanvas/init draw 1:1

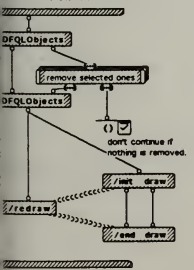


DFQLCanvas/end draw 1:1

DFQLCanvas/update 1:1



DFQLCanvas/remove DFQLObjects 1:1



▽ Manipulation Window

*Manipulatio..
▽
esme
NULL
▽
owner
FALSE
▽
active?
NULL
▽
window record
4
▽
def ID
FALSE
▽
modal?
FALSE
▽
close?
NULL
▽
selected item
[86 9]
▽
location
[448 592]
▽
size
* *
▽
activate method
"/Close"
▽
close method
"/ide proce..
▽
idle method
"/key press"
▽
key method
{ <<DFQLEdit..
▽
item list
0 needed for blinking
▽ time intervals
duration

Manipulation Window


open
int ivar
and open it

DFQLCanvas
click reset

DFQLCanvas
active when reset
button is pressed,
calls canvas' reset.

display inputter
displays DFQLEdit

hide inputter
hide it;
reset ivar
"selected item"

new DFQLObject
called from DFQLEdit
inputter when the <return>
is presssed

key press
process the key press;
key method of <self>

show SQL
Show SQL button click;
show the equivalent
SQL stmt

get relation
returns <Relation>
whose name equal
to the input parameter

refresh canvas
called by
activation
method

insert at cursor
inserts the selected text
from pop-up menus into
the current cursors position.

run query

idle process
blinks the objs in canvas
when there is an errror.

reset button
needed for activate method,
calls reset DFQLCanvas

delimiter
needed to activate method
insert at cursor by a button


help

goOutterLevel

goInnerLevel

convert DFQLObject

dose

incrNesting

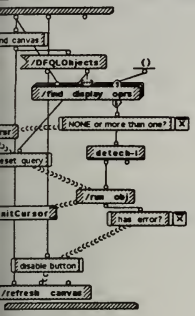
find display oprs

uncompact

compact

decrNesting

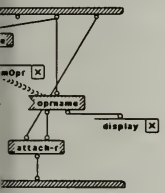
Manipulation Window/compact 1:2



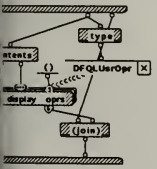
Manipulation Window/compact 2:2

ry is compacted. Uncompact first..."

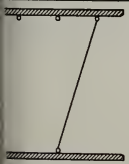
Manipulation Window/find display oprs 1:3



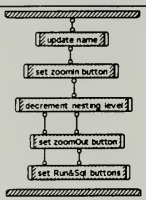
Manipulation Window/find display oprs 2:3



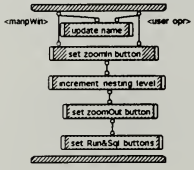
Manipulation Window/find display oprs 3:3



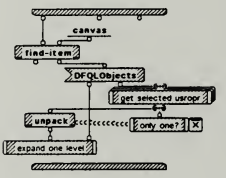
Manipulation Window/decrNesting 1:1



Manipulation Window/incrNesting 1:1

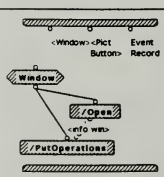


Manipulation Window/goInnerLevel 1:1

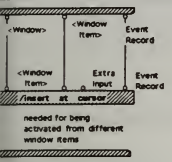


Manipulation Window/goOutterLevel 1:1

Manipulation Window/help 1:1

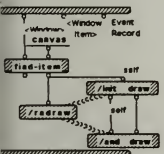


Manipulation Window/delimiter 1:1

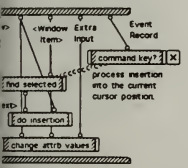


Manipulation Window/run query 1:1

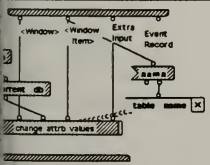
Manipulation Window/refresh canvas 1:1



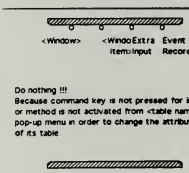
Manipulation Window/insert at cursor 1:3



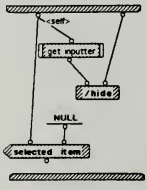
Manipulation Window/insert at cursor 2:3



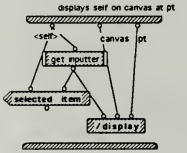
Manipulation Window/insert at cursor 3:3



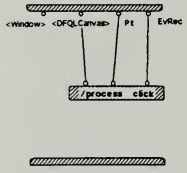
Manipulation Window/hide inputter 1:1



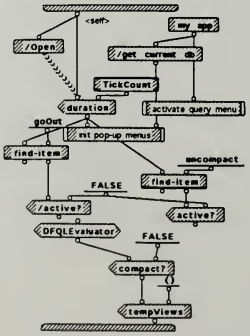
Manipulation Window/display inputter t:t



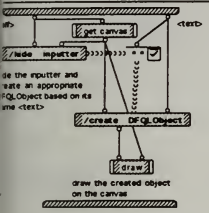
Manipulation Window/DFQLCanvas click 1:1



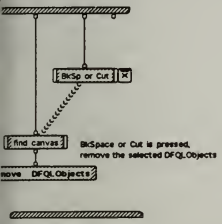
Manipulation Window/open t:t



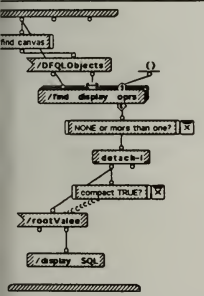
Manipulation Window/new DFQLObject 1:1



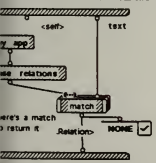
Manipulation Window/key press 1:1



Manipulation Window/show SQL 1:1

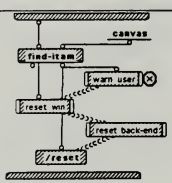


Manipulation Window/get relation 1:2

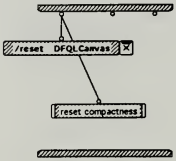


Manipulation Window/get relation 2:2

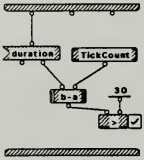
Manipulation Window/reset DFQLCanvas 1:1



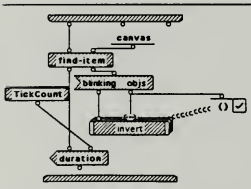
Manipulation Window/reset button 1:1



Manipulation Window/idle process 1:3

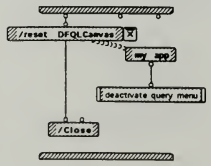


Manipulation Window/idle process 2:3

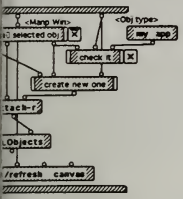


Manipulation Window/idle process 3:3

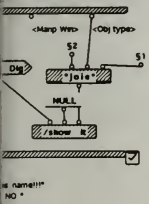
Manipulation Window/done 1:1



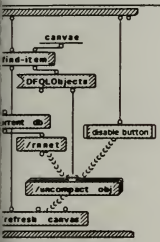
Manipulation Window/convert DFQLObject 1:2



Manipulation Window/convert DFQLObject 2:2



Manipulation Window/uncompact 1:1



INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
Professor C. Thomas Wu, Code CS/Wu Naval Postgraduate School Monterey, CA 93943-5002	2
LCDR John A. Daley, USN, Code CS/Da Naval Postgraduate School Monterey, CA 93943-5002	1
Kara Kuvvetleri Komutanligi Kutuphanesi Bakanliklar / ANKARA TURKEY	1
Kara Harp Okulu Kutuphanesi Dikmen / ANKARA TURKEY	1
T.C. Genel Kurmay Baskanligi Kutuphanesi Bakanliklar / ANKARA TURKEY	1

Mustafa Eser
Evsat Mah. Ozdilek cad. 11.Sok. No:6A
42700 Beysehir / KONYA
TURKEY 1

Top. Kd.Utgm. Turgay Cince
Sukraniye mah. Yuksel sk. No:34
BURSA
TURKEY 1

LCDR. Steve Sellner
26 Revere Rd. La Mesa.
Monterey, CA 93943 1